

Forest Electronic Developments WIZ-C MX

PIC Microcontroller Compiler

Manual for version MX 3 onwards

THE FED C Compiler (WIZ-C) 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 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.

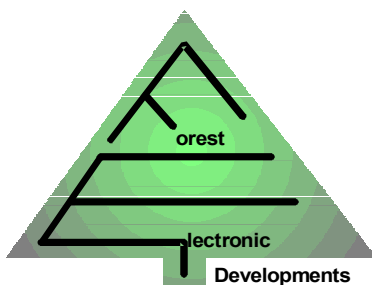
The professional version adds multiple project management and simulation and includes many additions to general simulation capabilities

The program, and its support files and example programs are © Copyright Forest Electronic Developments, 1998/2020

With thanks to Marcel Van Lieshout for good work on the pre-processor (2004)

The program may be installed onto the hard disk of one Personal Computer, and must removed by deleting the executable file, and all the support files before installing onto a different computer. The sole exception to this is where a single user installs on two machines (e.g. Work and Laptop) on which the program will never be used simultaneously.

Forest Electronic Developments
LYMINGTON
Hampshire
UK



info@fored.co.uk

Or see the **Forest Electronic Developments** home page:


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

Contents

- 1) Compiler Introduction
- 2) Tutorial
- 3) Special support for the PIC
- 4) Support for 3rd party compilers
- 5) Development Environment Reference Manual
- 6) Example Projects
- 7) Optimising Your Output
- 8) Using Assembler
- 9) Interrupts & Memory
- 10) Creating Libraries
- 11) Library Reference
- 12) C Reference
- 13) Pre Processor
- 14) Use with MPLAB
- 15) The Professional Version
- 16) Command line Interface
- 17) List of library functions
- 18) List of Keywords

Manuals provided with WIZ-C MX

The following manuals are provided with WIZ-C MX in PDF format :

Manual	C Compiler manual and introduction to WIZ-C, includes WIZ-C professional reference	
WIZ-C Introductory	Tutorials and reference for WIZ-C and the simulation system	
Simulation	Simulation and debugger reference manual	
Main Assembler Help & Manual Source	Reference for the Assembler and Linker used within WIZ-C MX	
Element Reference	Reference manual for the basic element set – certain elements such as the touch library have their own reference	
ICD Introductory	Reference manual for the PIC Key MX in circuit debugger system which operates with WIZ-C to program and debug applications on the real hardware.	

1 *WIZ-C and FED PIC C Compiler Introduction*

Information

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 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.

WIZ-C adds a drag and drop interface, WIZ-C includes all the functionality of the FED PIC C Compiler and any reference to the FED PIC C Compiler refers to WIZ-C equally.

The FED PIC C compiler will handle any of the current 14 bit PIC's and future devices may be added by changes to initialisation files which will be provided free of charge on our web site. All devices are handled by standard C header files.

Features :

- Designed to ANSI C Standards
- Integrated Compiler Environment
- Integrated Debugging Support
- Supports full range of 14 bit PIC Processors
- Efficient code production
- Wide range of library functions
- Includes floating point numeric support
- Includes fully integrated assembly level debugger
- Supports MPLAB

The compiler also includes full inline and infunction assembler support and is supplied with full manuals and examples on CD-ROM.

The enhancements which are available in The Professional Version allow the user to:

- Manage and simulate multiple projects together
- Connect PIC pins across projects to allow simulated devices to communicate
- Handle assembler and C projects
- View variables in native C format
- View a list of all local variables and their values
- Maintain a history within simulation to back track and determine the past leading up to an event

Please note enhancements available in The Professional Version are shown in the section at the end of this manual.

Contacting Forest Electronic Developments

FED may be contacted at:

Phone/Fax : 01590 - 681511

International : +44 - 1590 - 681511

Email : info@fored.co.uk

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

Bug Reports

Please reduce the program to the minimum possible size which still exhibits the error and submit the program with any other information you think may help to FED by post or email as shown above.

2 WIZ-C MX Tutorial

Tutorial Introduction

Start the Program and open a new project

Compile and complete the project

Assembling the code into a hex file

Initial Simulation

Check Serial receive routine

Examining local variables

Using the trace window

Checking a write operation

Checking a read operation on real hardware

Tutorial Introduction

In the example project we will look at the development and simulation of 2 complete programs using WIZ-C MX. This tutorial is included in the TutorCompleted folder in the documents folder of the installation, but you are recommended to run through the complete step by step project creation below if you are not familiar with the environment.

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.

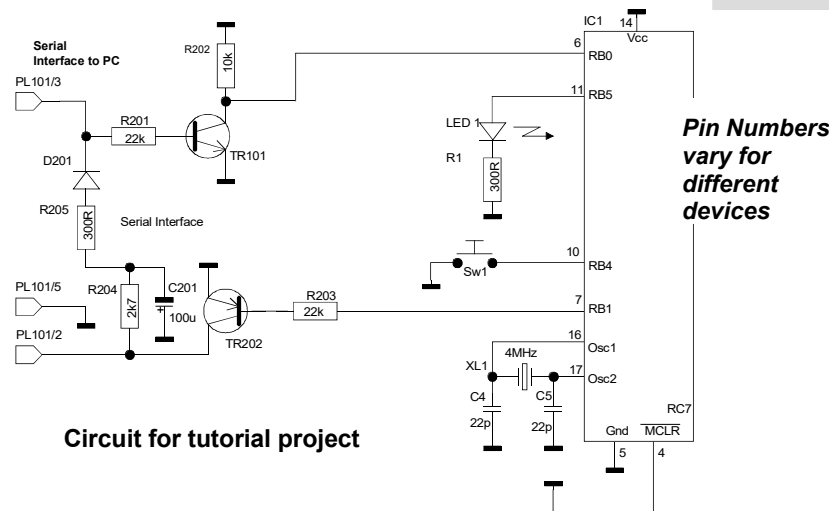
The program we will look at is designed for a 18F24Q10 processor. The application will undertake the following simple functionality:

- 1) It will have a serial interface which may be connected to a standard PC using a 9 pin socket (PL101), this will use the 18F24Q10 serial port.
- 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.

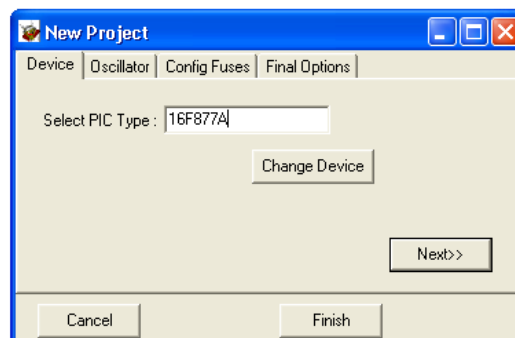
The circuit diagram is shown below:



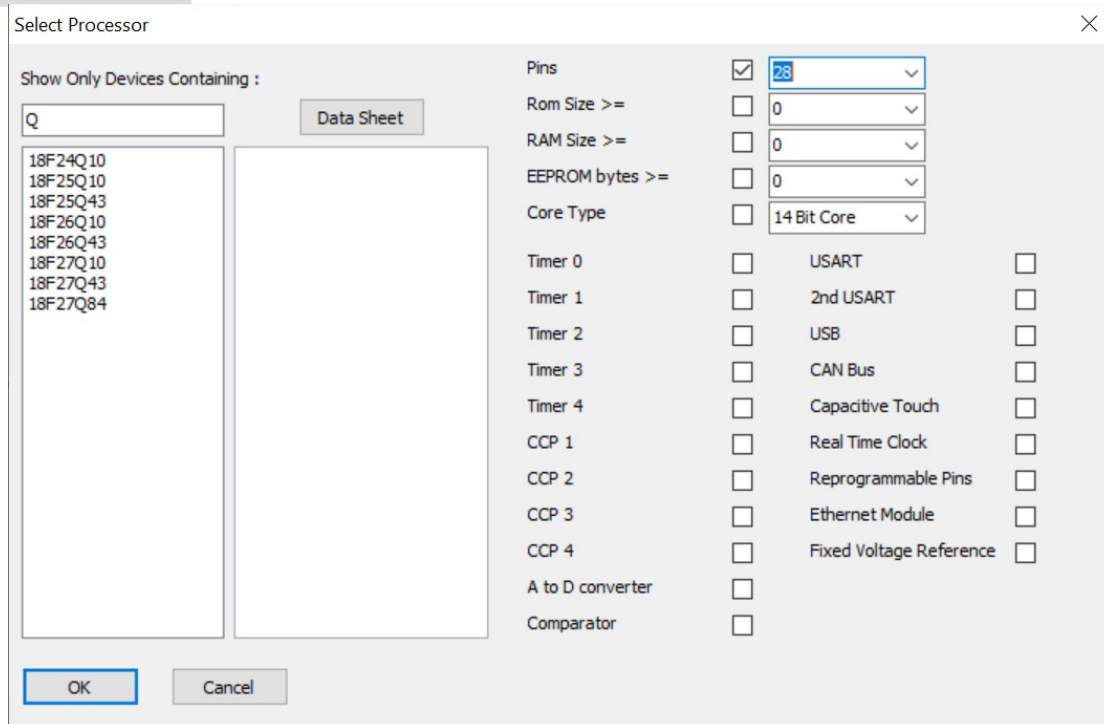
Note that by default WIZ-C will set Port B pull ups to enabled so there is no need to use a pull up on the switch input.

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 :



The default device is a 16F877, but we want to change that so click the Change Device button, a new dialog will be brought up – this is the Processor wizard, with this you can select the features you want – number of pins, ROM and RAM size and processor features. You can experiment with this – change the number of pins to 28, click the USART button and then you will have the following options :



Select Processor

Show Only Devices Containing :

Q

Data Sheet

18F24Q10
18F25Q10
18F25Q43
18F26Q10
18F26Q43
18F27Q10
18F27Q43
18F27Q84

Pins ☒ 28

Rom Size >= ☐ 0

RAM Size >= ☐ 0

EEPROM bytes >= ☐ 0

Core Type ☐ 14 Bit Core

Timer 0 ☐ USART ☐

Timer 1 ☐ 2nd USART ☐

Timer 2 ☐ USB ☐

Timer 3 ☐ CAN Bus ☐

Timer 4 ☐ Capacitive Touch ☐

CCP 1 ☐ Real Time Clock ☐

CCP 2 ☐ Reprogrammable Pins ☐

CCP 3 ☐ Ethernet Module ☐

CCP 4 ☐ Fixed Voltage Reference ☐

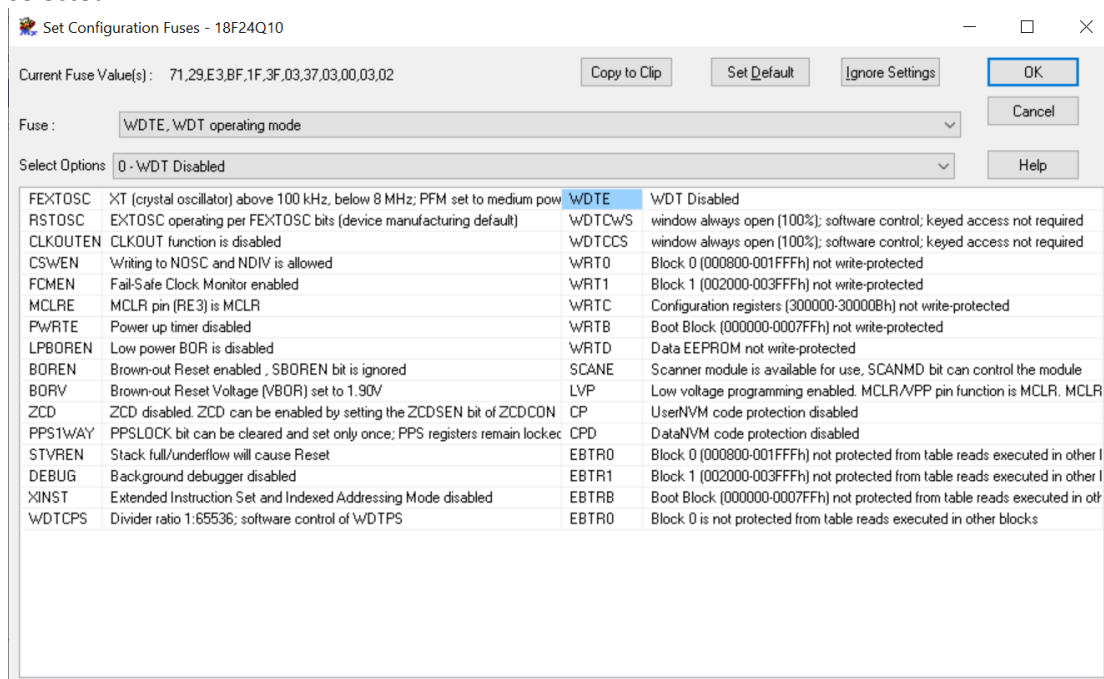
A to D converter ☐

Comparator ☐

OK Cancel

Type “Q” into the box headed “Show Only Devices Containing”, this will reduce the number of processors displayed. You can experiment with with the other options – for example clicking USB will only display devices which have USB hardware on board, and selecting a pin count will show only processors with the defined pin out.

Now select the 18F24Q10 device. Click the OK and then 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 – note the watchdog is disabled, an XT oscillator selected and the power up timer selected.



Set Configuration Fuses - 18F24Q10

Current Fuse Value(s) : 71,29,E3,8F,1F,3F,03,37,03,00,03,02

Copy to Clipboard Set Default Ignore Settings OK Cancel

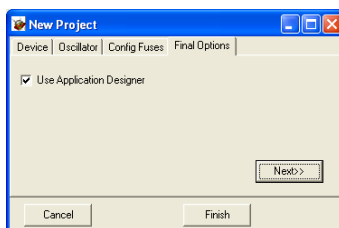
Fuse : WDT, WDT operating mode

Select Options 0 - WDT Disabled

FEXTOSC	XT (crystal oscillator) above 100 kHz; below 8 MHz; PFM set to medium power	WDT	WDT Disabled
RSTOSC	EXTOSC operating per FEXTOSC bits (device manufacturing default)	WDTCS	window always open (100%); software control; keyed access not required
CLKOUTEN	CLKOUT function is disabled	WDTCCS	window always open (100%); software control; keyed access not required
CSWEN	Writing to NOSC and NDIV is allowed	WRT0	Block 0 (000800-001FFFh) not write-protected
FCMEN	Fail-Safe Clock Monitor enabled	WRT1	Block 1 (002000-003FFFh) not write-protected
MCLR	MCLR pin (RE3) is MCLR	WRTC	Configuration registers (300000-30000Bh) not write-protected
PWRTE	Power up timer disabled	WRTB	Boot Block (000000-0007FFFh) not write-protected
LPBOR	Low power BOR is disabled	WRTD	Data EEPROM not write-protected
BOR	Brown-out Reset enabled, SBOR bit is ignored	SCANE	Scanner module is available for use, SCANMD bit can control the module
BORV	Brown-out Reset Voltage (VBOR) set to 1.90V	LVP	Low voltage programming enabled. MCLR/VPP pin function is MCLR. MCLR
ZCD	ZCD disabled. ZCD can be enabled by setting the ZCDSEN bit of ZCDCON	CP	UserNVM code protection disabled
PPS1WAY	PPSLOCK bit can be cleared and set only once; PPS registers remain locked	CPD	DataNVM code protection disabled
STVREN	Stack full/underflow will cause Reset	EBTR0	Block 0 (000800-001FFFh) not protected from table reads executed in other blocks
DEBUG	Background debugger disabled	EBTR1	Block 1 (002000-003FFFh) not protected from table reads executed in other blocks
XINST	Extended Instruction Set and Indexed Addressing Mode disabled	EBTR8	Boot Block (000000-0007FFFh) not protected from table reads executed in other blocks
WDTCS	Divider ratio 1:65536; software control of WDTPS	EBTR0	Block 0 is not protected from table reads executed in other blocks

OK Cancel Help

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 :

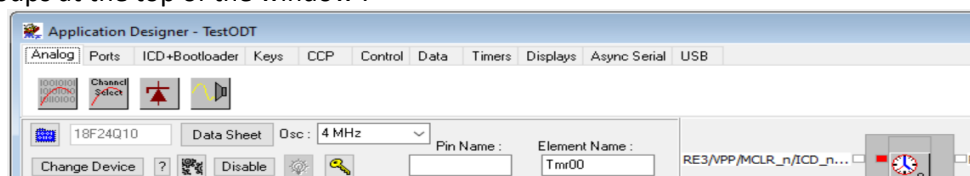


Finally press Next for the last time (or Finish).

A File dialog box will be brought up. Select a directory in which to save it. There is a folder called "Wiz-C Projects" in the user documents folder.

The top window will be the application designer. Check that the PIC type is 18F24Q10 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 :



Using elements within the application

The first element that we will select is the serial interface. Select the Async tab and hover over an element with the mouse - a small help box will appear with the element name. Select the element called "Interrupt Driven Serial Interface 1" by clicking it. The element icon looks like this:

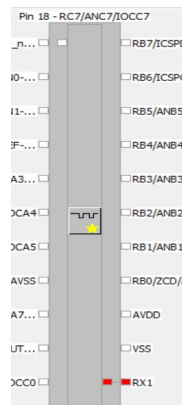


Now right click the element and a pop up menu will appear. Select the menu option "Help on selected element" and read through the help file entry for this element. It probably won't all make sense at the moment.

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. When you select some elements another element will also be included this other element been *hooked* into the project by the serial interface. In this case no element is hooked by the serial interface.

Note that the element for the USART reflects the real device in that the pins are remappable. To connect the element to the required pins on the device click the top pin on the small picture of the element, this will turn red, now click the pin RC7 on the picture of the PIC, Click the other pin on the element (TX) and connect to pin RC6.



Now we must set the parameters of the serial interface. Click the parameters tab and the parameters for this element will be displayed. Set the serial bit rate to 19200, Clear the box labelled “Use XON/XOFF protocol” – we don’t need this, finally leave the Receive and Transmit buffer rates at 32 and 8 respectively. It will now look like this :

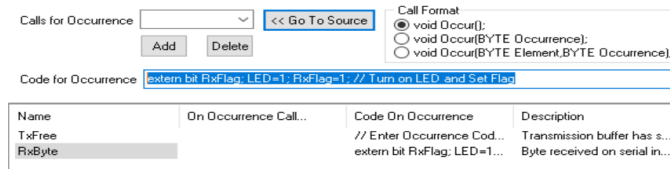


Some software elements including the Interrupt Driven Serial Interface element allow the user to define software functions to be called automatically when events occur. An event is described in the Applications designer as an *Occurrence*. In this case the Occurrence is that a byte has been received 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 are two occurrences, one for when the transmit buffer is free, and one for when a byte is received. When a byte is received we would like to call a C function called "LEDOn".

Click the second occurrence “RxByte” to select it. When a byte is received this element will either call a function or execute some code. We simply need to turn on an LED and set a flag when a byte is received, so in the box labelled “Code for occurrence” enter the following :

```
extern bit xRxFlag; xRxFlag=1; LED=1;
```

This should look like this :



Functions could be added here if wished and they would be called in turn if a byte is received. Note that we will need to define the external variable RxFlag elsewhere in the program and we'll do that later.

This completes the initialisation for the interrupt driven serial element.

You might notice that there is already an element on the designer. This picture of a timer with 0 is the Timer 0 element. This is automatically included in every application which uses the designer and the parameters are set to overflow at as close to 1mS as possible, this element is used by many other elements as a timer. For example in this case it is used by the push button element for debounce and repeat. Click on Timer 0 and look at the parameters. It is set to 1024uS overflow.



Now we can add the switch which will be connected to pin B4. Select the Keys tab and double click the "Simple Switch" icon:



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, click pin RB4 to connect 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 a number of symbols defined in the header file which can be referenced anywhere in the programme - as follows for this example :

```
#define TxSwitchPort PORTB
#define TxSwitchLat LATB
#define TxSwitchBit 4
#define TxSwitchBitMask 16
#define TxSwitch bRB4
#define TxSwitchTris bTRB4
#define TxSwitchOut bLATB4
```

We can test the input directly using the definitions as follows:

```
if (TxSwitchPort&(1<<TxSwitchBit)) {..// DO Something
```

alternatively we can test it using the bit variable which is created with the same name :

```
if (TxSwitch) {..// DO Something
```

Examine the parameters of the switch - we want an auto repeat so click the Repeat option so that when the button is held it will automatically repeat. 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" into the box "Calls for Occurrence" and then click Add to include it to the list of calls for the occurrence SBPress0. Now when the button is pressed the function TxLastRx will be called. This is slightly different from the occurrence when a byte is received on the serial interface – in that case we included a snippet of code to be executed when a byte was received, here we have a lot more work to do when the button is pressed so we create a function – TxLastRx – to handle the action.

Now we can add the LED output which is on pin B6. Click the Ports Tab at the top and add the "Port Driver" element:



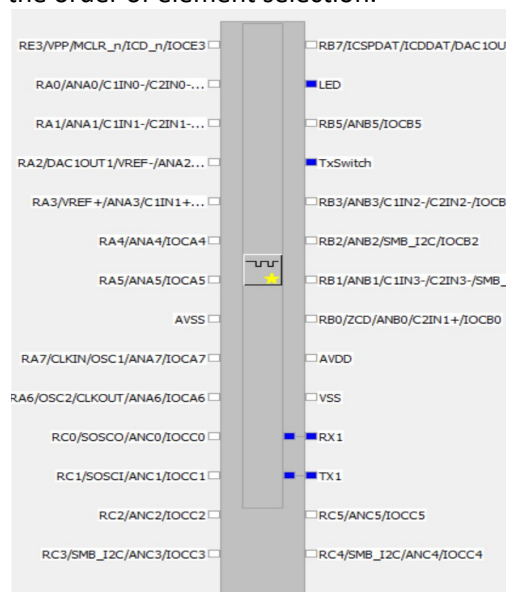
Connect the port output to pin RB6 and name it "LED". Initially the LED will be off so under the parameters tab select the initial value 0.

Finally we need to make all of the pins of 18F24Q10 digital I/O pins instead of some being Analogue input pins. Add the first element from the Ports tab which is the All Digital element :



This has no parameters or pins or occurrences, it simply makes all analogue input pins set to digital I/O pins.

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 Device Picture to Clipboard* option of the same menu.

Check the bottom of the application designer as well to make sure you have included all the elements :



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 with errors as 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:

```
extern bit xRxFlag; // 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:

```
xRxFlag=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 function that should be called when an occurrence happens should have been entered as templates, if they don't exist then enter manually as follows :

```
void TxLastRx()
{
}
```

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 AddTx 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. Note the WaitRx() function – this picks up the next byte from the receive buffer – we need this to detect the received byte. This information is used to help finish the second routine which will transmit the last received value when the button is pressed. Now go back to the edit window and change the function as follows:

```
void TxLastRx()
{
    if (!xRxFlag) return;          // Return if no byte received
    xRxFlag=0;

    // void pSerialOut(unsigned char Tx);
    // - Transmit value to port
    AddTx(WaitRx());

    LED=0;          // Turn off LED
}
```

The complete Tutor_User.ASM file should now look like the listing below (the include file may vary according to where you saved the file). Use the *File / Print* menu option to print the file if you wish. Check the file and edit any changes.

```

#include "D:\\Data\\CDev\\Projects\\CCompiler\\PIXIE\\Bugs\\Tutor_Auto.h"
#include "Tutor.h"
bit xRxFlag;

//
// 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()
{
    xRxFlag=0;
}

//*****
//
// 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 (!xRxFlag) return;          // Return if no byte received
    xRxFlag=0;

    // void pSerialOut(unsigned char Tx);
    // - Transmit value to port
    AddTx(WaitRx());

    LED=0;                          // Turn off 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 PIC18F24Q10 and run directly – the configuration fuses have been set automatically as selected at the start of the project. However we can also simulate it and look at the results on simulated devices and the waveform analyser.

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.

Switching screen layouts

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

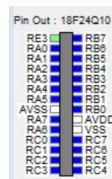
Compact	Press ALT+C keys
Debugging	Press ALT+D keys
Editing	Press ALT+E keys

In compact mode all windows (Debug, Project, Editing 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.

Simulating with external devices

Press ALT+D or ALT+C to switch to the debugging or compact layouts. 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, note the white pins which are not on I/O ports and the blue pins which on reset are Analogue inputs.




WIZ-C has the capability to simulate LED's, switches, LCD displays, Serial and USB devices and a number of other devices which might be connected to the PIC. Devices which are not simulated directly (such as CAN) can be emulated by creating injection files which insert defined values at specific points in the simulation.

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. The LED model is an RGB device, you can set it to be any colour, or combination by connecting the Anode and Cathode of each of the three LED's to the same, different, or no connections. Under the Connections box there is a list box called Pins with two entries "Anode" and "Cathode" for Red, Green and Blue LED's. Select Red Anode and then use the Port box to select Port B, use the Bit box to select bit 6. Select Red Cathode and click the Connect Low option. This will connect the LED between Bit B6 and ground and make it light Red when powered. Press the OK button.


Note how the LED appears on the debugging window, you can drag it around to a convenient location.

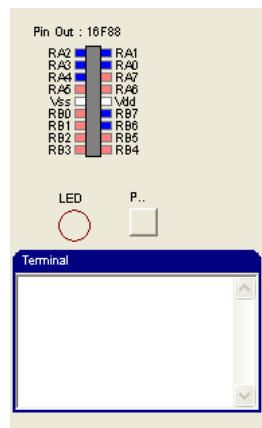
Next the push button. This time we'll use a short cut.

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:



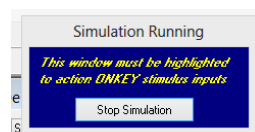
A push button will be created, you don't need to set up the connections as . Do the same for the serial element – click the element and then click the Create Device button.

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:



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 RB6 output changes from dark red to light red as the LED goes on and off. Stop the program by pressing the stop bottom on the Simulation Running box :



Change the TxLastRx() function as follows :

```
void TxLastRx()
```

```

{
  if (!RxFlag) return;          // Return if no byte received
  RxFlag=0;

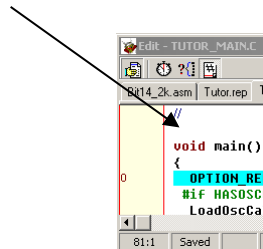
  // void pSerialOut(unsigned char Tx);
  // - Transmit value to port
  while(GetRxSize())
  {
    AddTx(WaitRx());
  }

  LED=0;          // Turn off LED
}

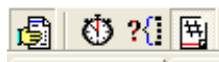
```

Compile and run the program again, this time type several characters into the terminal window, press the button, note how they all the characters are output correctly in order, this shows the correct operation of the serial buffer.

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 :

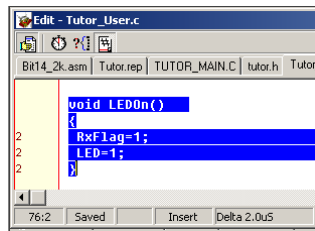
```

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.

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".

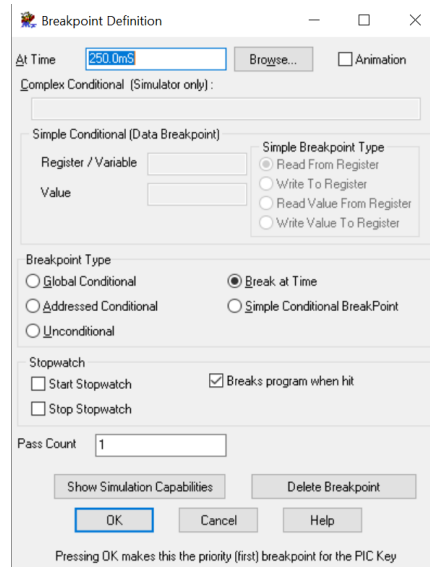
Select the project window and press Insert, you can now select the file which has just been created (you may need to select "Files of Type" *.sti. A dialog box will open. Select Stimulus under the "Files of Type" box. Press OK. Check that the file type is "Stimulus" in the Project File dialog box and press OK. Double click the file "TutorInput.STI" in the project window. We will generate a stimulus file which enters the character 'A' then 'B' at time 50mS, and then the key is pressed at time 100mS. To do this enter the following information (again the Simulator introductory manual covers this in more detail).

```
50m
serial19200-PORTC:7='A'    ; Send character A at 50mS
+2m
serial19200-PORTC:7='B'    ; Send character B 2mS later
100m
PORTB:4=0                  ; At 100mS press the button
+100m
PORTB:4=1                  ; And leave
```

Save the file by using the *File / Save All* menu option. Now we want to run the program until time 250mS and then stop to check the inputs and outputs. Bring up the breakpoint window by clicking the arrow on the minimised window:



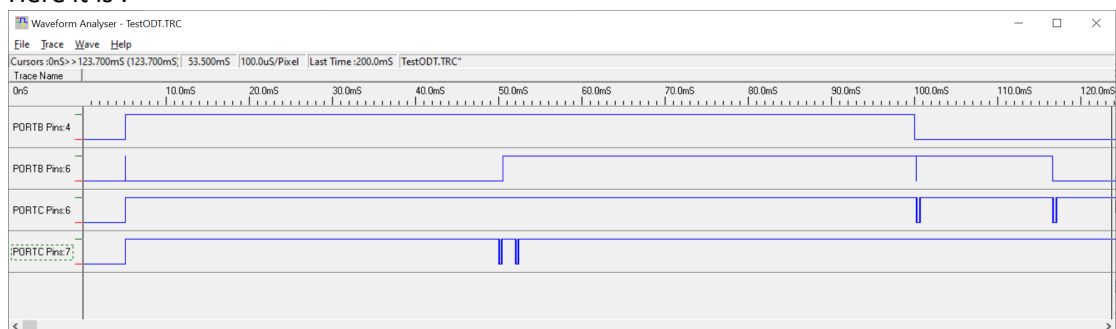
Right click the window and use the "Add Breakpoint" option. The Breakpoint Definition dialog box will be shown. Select "Break at Time", and then in the At Time box enter 250mS, click the OK button to define the breakpoint:



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 250mS. 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". Then do the Same with Port C, you can click on unwanted traces and delete them. We only need to look at B4, B6, C6 and C7. Resize the window to a comfortable size.

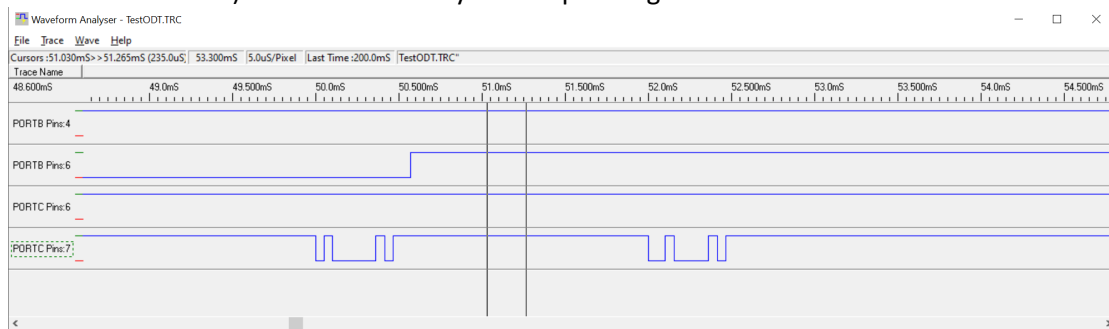
Finally press the F8 repeatedly button to zoom out and watch the received byte on Port B, bit 2, followed by Bit 6 going high (the LED turning on). Then at about 21mS the key press is detected (remember it is being debounced and takes a few milliseconds to detect it) – the LED goes out and the byte is transmitted back on the interface. You can see the received byte being transmitted on Port B, bit 5 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 can zoom up around the Bytes received by dragging the cursors (vertical lines at each end of the window) around the two bytes and pressing F6:



You are recommended to run through the simulator manual which follows this introductory manual.

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 just a few lines of code. Although in itself this project may not be that useful, it can operate in any project to provide subsidiary timing functions.

The project uses an older device – the 16F877A, though it would be sensible to change to a more modern (and cheaper!) device if used in reality. The project is in the DigClock folder of the Documents folder – “WIZ-C Projects” then “Sample Projects” folder.

We won't go through the step by step creation of the project – open it and have a look at how it's constructed.

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.

To set the time these variables should be updated noting that they should be in range, so that if Min's is incremented and reaches 60 then it should be reset. We can see this in the example.

The accuracy of the clock is related to the use of a crystal which is a power of 2, so that a common crystal of 4.096MHz, or 16384MHz should be used. A 4MHz crystal may still be used, but will have an error rate of 50 seconds per day. The element used adjusts automatically to the crystal rate.

For this example we shall operate a 24 hour clock showing hours, minutes and seconds, days will not be displayed. There will be three push buttons to set the time - set minutes, set hours and set days.

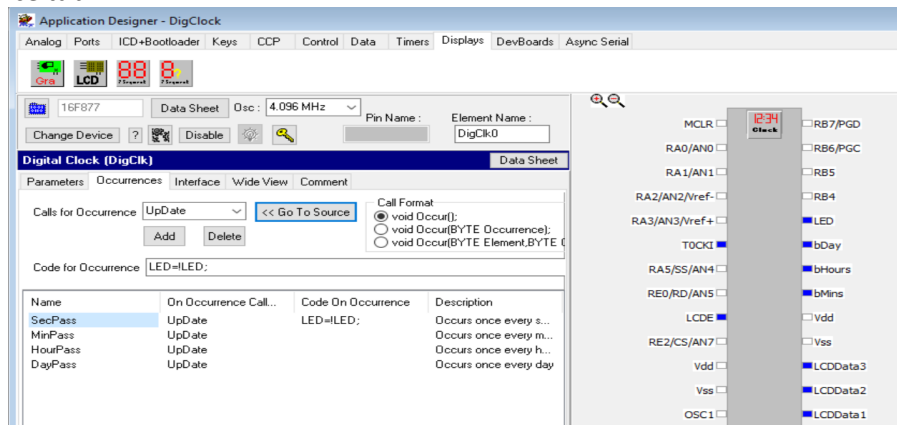
LCD Displays

Digital Clock

The digital clock element looks like this:

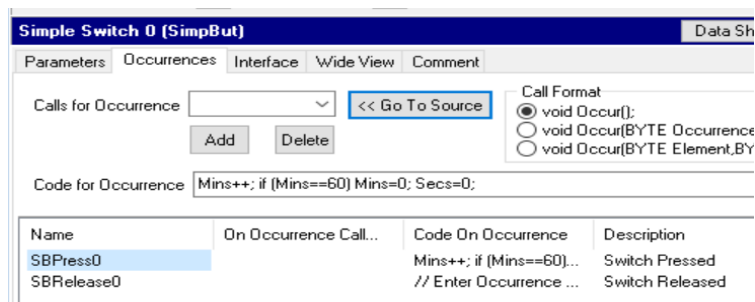


The only parameter is 12 or 24 hour selection. For this clock we'll use 24Hour. Now click on the Occurrences tab. Every time that the time changes (seconds, hours or minutes) then we'll update the display. Note that the UpDate function is called for each occurrence. To flash the LED the SecPass occurrence includes the code "LED=!LED". Note that on the occurrence tab:



Switches

Have a look at the occurrence for the three switches. You can see that each updates one of the variables, checks it is not out of range, the example below shows that when the Mins button is pressed then the variable is updated and checks that if it reaches 60 then it is reset. Note that the Seconds variable is set to 0 every increment of the minute variable so that the exact time may be set.



User Code

Open the DigClock_User.C file. A welcome message is set on power up and left in place for 2 seconds:

```
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. CPrtString is the function that we can use, however it does not print a leading zero if the number is less than 10.

```
//
// 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 the UpDate() function which will print the time to the display. At 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.

Note that in this example a non-buffered LCD is used, this uses slightly less RAM, but needs the RW pin to be connected – you can see this on the LCD element.

Final application

Generate the application (use Ctrl+F9). The project may be simulated using the external devices. Watch the display and press the hours and minutes buttons. Note that the simulation is usually faster than real time owing to the use of a 4.096MHz clock.

Including the day

Adding the day was very straightforward. A new push button element was 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. Note the use of a string array in ROM to determine the day to be printed.

```
//
// 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. If the token “const” is removed then it will be stored in RAM using RAM space as well as initialisation time. Try it to see the effect.

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.

3 *Special support for the PIC*

Header files
Port Bits
Register Bits
Port Structure
Macros
Memory Allocation
EEPROM Support
Large Programs
Configuration Fuses
Extended Instruction Set

Header files

The simple method of including header files for the PIC is to use `pic.h`. This will automatically include the correct header file for the PIC selected in the project options. For example:

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

This include file also includes `PortBits.h` and the relevant register bits file for the processor (see below).

If it is desired to fix the PIC type then there is a header file for every type of PIC supported by the FED PIC C Compiler. The header files are called "P1nnnnn.h" where nnnn is the processor type, e.g. `P16F84.h` or `P18F452.h`. The advantage of including the header file like this is that an error will be generated if the wrong PIC is selected in the project options for the header file defined.

```
#include <P16F877.h>
#include <P16F877_bits.h>
#include <PortBits.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. For every register there is defined a structure which includes all the bits in the register. This is named "register_namebits", e.g. `ADCON0Bits`.

So to set the GO bit in `ADCON0`, and then wait for AD conversion to be complete use :

```
ADCON0Bits.GO=1;
while (ADCON0Bits.GO) ;
```

For many registers this is a union of several groups of bits which allow different names for each bit. It is wise to examine the header file for a register to see how it is defined.

For example consider `ADCON0` on this device. The register is defined as follows,

```

extern unsigned char ADCON0; // Register

// - ADCON0
const int GO_DONE=2; // Bits in the register defined as constants
const int nDONE=2; const int GO=2;
const int CHS0=3;
const int CHS1=4;
const int CHS2=5;
const int ADCS0=6;
const int ADCS1=7;
const int ADON=0;
const int GO_nDONE=2;
const int CHS=3;
const int ADCS=6;

// Register : ADCON0 // Union defining bits in the register
typedef union { // Note this is the microchip method of addressing
struct
{
    unsigned:2;
    unsigned GO_DONE:1;
};
struct
{
    unsigned:2;
    unsigned nDONE:1;
};
struct
{
    unsigned:2;
    unsigned GO:1;
    unsigned CHS0:1;
    unsigned CHS1:1;
    unsigned CHS2:1;
    unsigned ADCS0:1;
    unsigned ADCS1:1;
};
struct
{
    unsigned ADON:1;
    unsigned:1;
    unsigned GO_nDONE:1;
    unsigned CHS:3;
    unsigned ADCS:2;
};
} ADCON0bits_t;
extern ADCON0bits_t ADCON0bits;

extern bit bGO_DONE; // In P16F877_bits.h, bit definitions
extern bit bnDONE;
extern bit bGO;
extern bit bCHS0;
extern bit bCHS1;
extern bit bCHS2;
extern bit bADCS0;
extern bit bADCS1;
extern bit bADON;
extern bit bGO_nDONE;

```

For example to enable interrupts any of the following lines could be used

```

INTCON|=0x80; // Direct setting of bit
INTCON|= (1<<GIE); // Enable interrupts
INTCONbits.GIE=1; // Through the structure
bGIE=1; // GIE defined as a bit

```

Which is preferred is up to the coder. The Microchip preferred method is through the structure. The bit method is shortest. All use the same code - this type of C code is translated into PIC BSF and BCF instructions, and is, therefore, efficient.

Port Bits

There is a header file called PortBits.h. It is included automatically by pic.h.

Include this file to define the port bits in the form

```
bRA0, bRA1... bRB0... bRE1
```

etc.

Similarly the tris bits are also defined as :

```
bTRA0, bTRA1... bTRB0... bTRE1
```

etc.

The following code places bit 0 of PORT B into the driving state and then turns on bit 0 :

```
#include <PortBits.h>

bTRB0=1;          // Drive
bRB0=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:

```
#include <PortBits.h>

x=bRC4;           // Test bit 4 (return 0 or 1)
```

For compatibility with compilers which only define the register bit as a bit not as a number, there is an option to define the bits without the b in front of them – see [Support for 3rd party compilers](#)

Note that the bit type used is non-ANSI and so will not be portable to all compilers.

Register Bits

Each processor has a header file which defines all the bits in special file registers as bit types. This file is automatically included by including pic.h.

Each bit is the same name as the normal bit name with a b in front of it. For example the GIE bit has a bit type called bGIE :

```
bGIE=1;           // Enable interrupts
bGIE=0;           // Disable interrupts
x=GIE;            // Constant GIE has the value 7 (the bit number)
```

For compatibility with compilers which only define the register bit as a bit not as a number, there is an option to define the bits without the b in front of them – see [Support for 3rd party compilers](#)

If it is desired to fix the PIC type then there is a header file for every type of PIC supported by the FED PIC C Compiler. The header files are called "P1nnnnn_bits.h" where nnnn is the processor type, e.g. P16F84_bits.h or P18F4520_bits.h.

This file may be included using #include, eg.

```
#include <p16f84_bits.h>
```

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)
```

Macros

The following macros are provided in pic.h :

bitset

```
bitset(reg,bit)

e.g.

bitset(PORTB,8);
```

This macro will set the specified bit in the specified variable.

bitclear

```
bitclear(reg,bit)

e.g.

bitclear(ADCON0,ADEN);
```

This macro will clear the specified bit in the specified variable.

Memory allocation

Memory is allocated automatically for all global variables. The most frequently accessed variables are held in the lowest memory pages.

You can manually allocate memory for large arrays, or when it is necessary to know where an array is to be stored – for example when working in conjunction with assembler. The following example shows how to allocate memory to a large array:

```
int GreenScore;           // Allocated by compiler
int RedScore;             // Allocated automatically
extern int Individual[32]; // Array of integers
#pragma locate Individual 0x120; // Locates array at 0x120
```

It is possible to force a variable into page 0 of a 14 core item, or to the bottom access bank of 16 bit (18 series devices). To do this use the register keyword :

```
int register GreenScore;           // In lower RAM page
```

This is useful when assembler routines need to assume a page for efficiency, or when frequently used variables need to be the most efficient. For example using 18F devices you can use the access bank, and not have to worry about setting the BSR register if you are addressing register variables. With devices which have an access bank which is not at address 0 (for example the 18F47K42 which has the access bank at 0x500), then the compiler still places the register variable in the access bank.

Be careful with register variables – if too many are defined the compiler will simply allocate them back into higher ram pages, this will not cause any problem with the compiler, but may cause problems with assembler routines, including some of the FED library routines. In general between 50 and 100 bytes of register variables (dependant on processor family) may be defined before they are pushed into higher pages.

EEPROM Support

Support for EEPROM data is provided by #pragma extensions to the C language :

```
#pragma eeprom data,data,address=data,data...

or

#eeprom data,data...
```

The EEPROM data is stored in the hex file and loaded by the simulator on reset.

Use either #pragma eeprom or #eeprom at the start of a line to introduce EEPROM data. This is followed by any number of bytes to be written in turn to the data area. If an item is of the form :

```
Address=data
```

then the data is written to the supplied address and all following items will be written to consequent addresses.

For example

```
const int val=0xb6;

#eeprom 0=1,7=8,9,10,val
#pragma eeprom 16=8
#eeprom 0xf7
#eeprom 0x78,99
```

In this example the first 32 bytes of EEPROM data will look like this :

EEPROM[00-0F]	01 FF FF FF FF FF FF 08 09 0A B6 FF FF FF FF
EEPROM[10-1F]	08 F7 78 63 FF FF FF FF FF FF FF FF FF FF

Note that the simulator loads the EEPROM data from the hex file on program reset, and it can be seen on the “Special” tab in the debugging window.

Creating new devices

FED auto create new devices using the Microchip standard data files, these are updated very quickly and new devices can be generated on request if not in the current installation.

Large Programs

Note that for processors with more than 32K words of program the standard 2 byte pointer can only point to the bottom 32K of ROM memory. To allow pointers to address functions in this situation any function which may be called through a pointer on a device with more than 32K ROM should be defined as pointed :

```
void pointed MyCallableFunction(int param);
```

The function will be placed in the bottom 32K words of ROM.

Therefore there is a limit imposed on the program that no more than 32K of functions may be called through a pointer. In practice this limit is far greater than required - if more than this size of program is required then small functions can be defined which call bigger functions.

There is no need to use the pointed keyword for processors or programs which utilise less than 32K words of program memory.

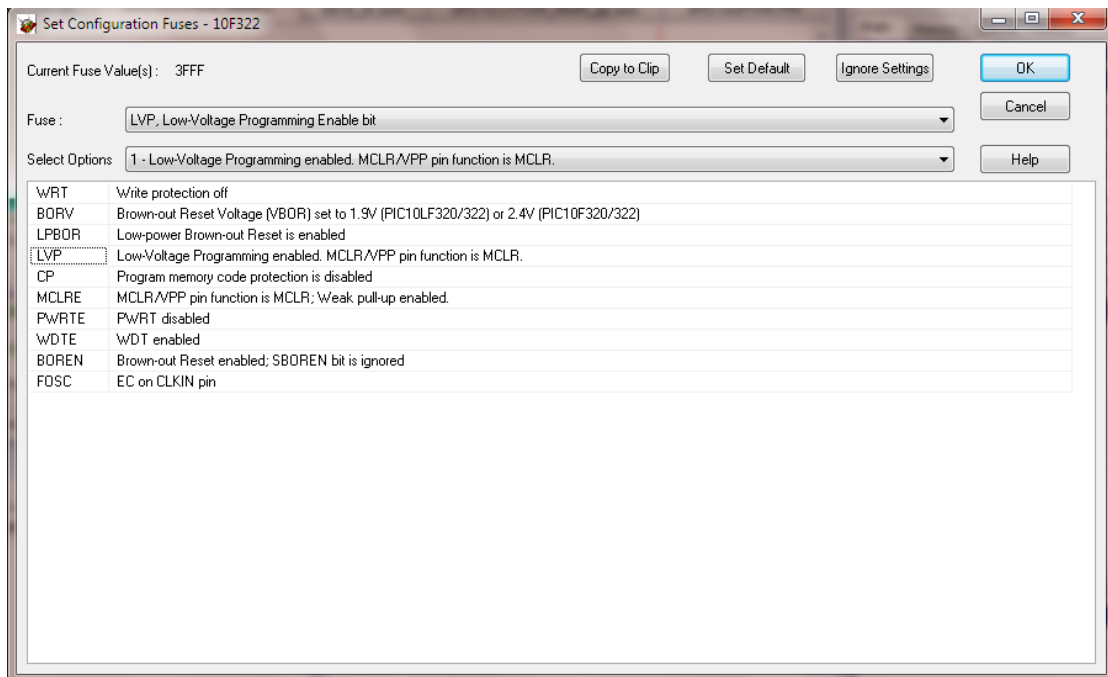
Configuration Fuses

The C Compiler provides support for simple set up of the configuration fuses.

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 :



The Current Fuse Value(s) line shows the hex value of the configuration fuse (or fuses if there are more than one).

The Set Default button will return all the fuse values to the unprogrammed (erased) value.

The Ignore Settings Button will close the dialog box and will force the compiler to ignore any settings made in this box. The fuses will be left unprogrammed, or will be read from the source files if present.

The Copy to Clip button will copy the fuses to the clipboard in a form that they can be read in the program.

To change a fuse value select the fuse from the first (Fuse) drop down box. The second ("Select Options") box will then show all the options available for that fuse when it is selected. Click this box to select the desired option for the fuse. Alternatively click a cell in the table to select that fuse and allow its values to be set.

When the processor type is changed from the application designer (WIZ-C only), or from the Project | Options dialog box the configuration fuses are then ignored. They must be set using the Set Configuration fuses box again before they will be included in the hex file.

#__config statements in the main source files take precedence over this dialog box. For example the following statements included in a C file set the config fuses as shown :

```
#ifndef __18F14K50
  __config _CONFIG1L,0x00
  __config _CONFIG1H,0x32
  __config _CONFIG2L,0x06
  __config _CONFIG2H,0x1e
  __config _CONFIG2H+1,0
  __config _CONFIG3H,0x08
  __config _CONFIG4L,0x01
  __config _CONFIG4L+1,0
  __config _CONFIG5L,0x03
  __config _CONFIG5H,0xc0
  __config _CONFIG6L,0x03
  __config _CONFIG6H,0xe0
  __config _CONFIG7L,0x03
  __config _CONFIG7H,0x40
#endif
```

Note this method works equally well with devices which store configuration fuses at the top of ROM rather than a non-volatile config fuse space. (An Example is the 18F87J50).

Extended Instruction Set

The compiler supports the use of the 18 series extended instruction set. This is normally disabled as the FED compiler Local Optimisation space makes the extended instruction set less efficient than the normal optimisations.

However re-entrant code which cannot use the "Function Parameters can be global" optimisation may benefit from the extended instruction set.

To use the Extended Set clear the box in the Compiler Options, Optimisation tab "Do not use extended instruction set if available". The Configuration Fuses will be set automatically to allow the use of the instruction set to allow simulation and the real program to run correctly. It is always worth checking that the fuse is enabled correctly in the programmer.

4 Support for 3rd party compilers

Introduction

Notes on incompatibilities

Introduction

With WIZ-C MX support for 3rd party compilers has not been upgraded to the most recent capabilities – FED recommend that source code is manually converted where possible if the most recent upgrades are assumed.

The FED compiler supports 3rd party compilers in that it is possible to compile source files designed for these compilers. Normally C source can be compiled directly although embedded assembler may require some work. Most special syntax is supported directly although not documented here, for example the FED #asmend directive may be replaced by #endasm in source files and both will work fine. The latter form is provided for compatibility with 3rd party compilers.

Special headers are provided which define macros to allow compatibility and these should be included at the top of each source file:

The files are:

hitec.h for files intended for the HiTec compiler

ccs.h for files intended for the CCS compiler

An example of their use is shown below:

There are a number of extensions to the C syntax provided for 3rd party compatibility. These are not fully documented here as they are all duplicated by FED C constructs and there is little point in providing full documentation for several methods of achieving the same end. However they are summarised below:

#pragma interrupt_level Sets the interrupt level of the following interrupt function.

@ Use of @ symbols to locate variables.

interrupt Keyword used to define a function as an interrupt function

persistent Keyword which has no function in FED PIC C compiler

Compatibility Options

The Compiler Options Dialog Box has a tab for compatibility options. To show this dialog then use the **Project | Set options for Project** menu option. This tab includes a number of features which it is desirable to disable unless a specific compiler syntax is to be supported.

There a number of buttons on the tab to set the default for the compatibility options for the particular target compiler. Click the button to set the options for the compiler required.

The types int1, int8, int16 and int32 are provided for 8 bit, 16 bit and 32 bit integers (identical to FED PIC C compiler types bit, char, int and long). These are used for internal conversion where compilers have a different bit length and are referenced in the header file.

The options are as follows:

_ in front of asm names. This option forces the compiler to create a macro for each C variable with the C variable name with an _ in front of it. For example if the C variable ix is defined then in assembler it may be referenced with the name ix or _ix when this option is selected.

This DOES NOT apply to internal register names such as STATUS which must be defined in assembler without the underscore. Conversion must be undertaken manually.

No 'b' _ in front of bit names. This option forces the compiler to define a macro value _BITTYPES_OPT. This will then define bit names for register without the 'b' in front of them. For example the bit T0IF is defined in the FED compiler as bT0IF. If this option is selected then the bit may be addressed either as T0IF or bT0IF.

Automatically include Header. This selections allows entry of a header file name which will be included automatically by pic.h. The header file usually includes macro replacements to substitute 3rd party function names for FED function names. Normally this is set by automatically clicking the button for the compiler with which compatibility is required.

Notes on incompatibilities

Assembler

The most likely source of problems is likely to be complete assembler functions. The FED compiler includes some options to assist with assembler – for example the option to prepend an underscore before C variable names, however some work may be required to convert assembler. See [Using Assembler](#).

Assembler blocks and comments

Some compilers mix C and assembler syntax in assembler blocks. FED C does not allow this. For example // to comment in assembler files may be allowed by 3rd party compilers, this should be replaced with a ; character as normal for assembler.

Converter programs

Some 3rd party compilers require a conversion program to be run in advance of the main program – the converter program will create a new C file called xxx_conv.c where xxx is the original file name. FED recommend that the converted file should now be worked on and the original file discarded. At present the following conversion programs are supplied within the converters sub-directory :

ConvertCCS.exe	To convert CCS programs in advance of compiling with the FED compiler.
----------------	--

Interrupt function

The FED C compiler operates interrupt functions differently from other compilers, there being no consistent approach. See the manual section [Interrupts & Memory](#).

Existing interrupt code may require re-writing, or the code may be moved outside the interrupt and tested with a flag set inside the interrupt. Some compilers support a named function for each interrupt and these will need to be converted to code inside the main interrupt function.

Untyped functions

The FED C compiler does not allow for untyped function declarations. The following is illegal :

```
GetADC()  
{  
}
```

Please replace with the following :

```
void GetADC()  
{  
}
```

5 *Development Environment Reference Manual*

Project

Compiling a project and reviewing errors

On-line help

Project Archiving

Menu Commands

Windows

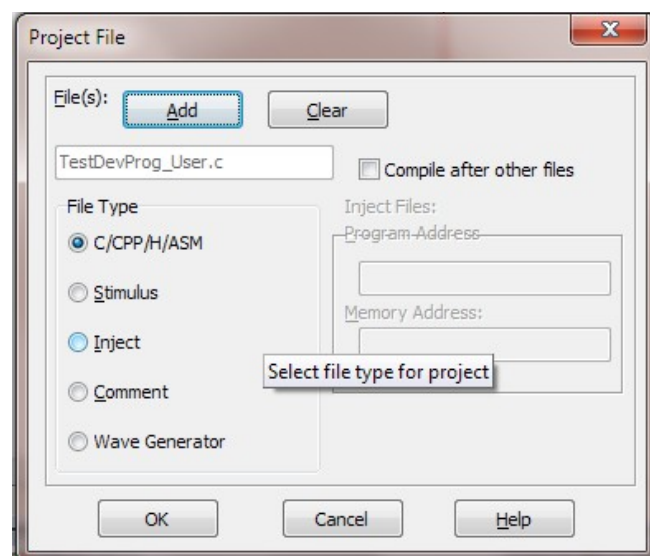
Please note enhancements available in The Professional Version are shown in the section at the end of this manual.

Project

A project is the name for the collection of files. These files are tagged to show that they will be included in the compiler process, or that they are text files which may document aspects of the project.

Those files which are tagged as C or assembler files are compiled to a binary object file. Simply add all the files which make up the program into the project window, and when the make command is used they will be assembled in the order in which they appear in the window.

When you add an item (or items) to the project a dialog box is shown which allows the file type to be selected. These types are C/CPP/H/ASM, Stimulus, Inject, or comment. The Stimulus and Inject file types are used with the simulator and are shown in the simulator help file. The C/ASM type of file is included when the project is compiled and assembled, the file extension is used to decide if it is a C or assembler file. The comment type is not used by the compiler, the assembler or simulator, it is normally used for “readme.txt” files, or other help associated with the project.



To create a new project then use the **Project | Open/New** menu option and type in a name for a project which does not exist. (Note for the professional version the menu option will refer to a project Group – for now it does exactly the same). Now use the **Project | Add Item** menu option (Professional Version **Project | Add/Insert Item to Current Project**), (or select the project window and press insert) to add the first file to be assembled. This will normally be an include file. Now you must create assembly files to make up the program. Either open existing files, or use the **File | New** menu option to create a new file and then use **File | Save As** to save it with a new file name. Finally use the **Project | Add item** menu to add it to the list of files in the project. If the **Project | Add** command is used on a file that does not exist then it will be created, double clicking the project item will load it into the editor window.

Projects are saved automatically when a new project is opened, or when you exit PICDE. When you re-enter PICDE the last project that was used is automatically opened. A project file has an extension of .PIC, and it saves the position, size and type of all the windows that were open when it was saved. This allows users to continue work from where it was left.

The project window shows a list of all the files which are compiled and linked for the current program. Note that C, Assembler, Stimulus/Injection and comment files are included under different tabs in the project window. Double click on a file in the project window to open that file for editing. To add and delete files use the commands on the project menu.

Compiling a project and reviewing errors

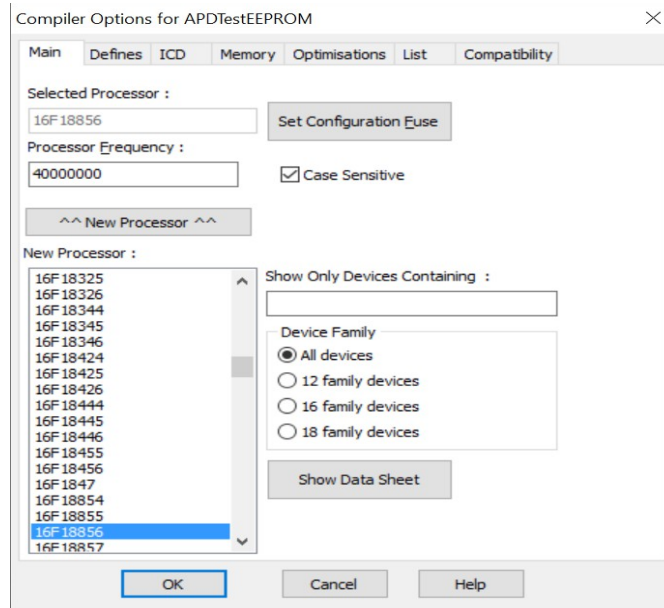
To compile a project then use the command on the button bar, or use the menu option **Compile | Generate Application**. For the very first time that the project is compiled the Compiler Options Dialog Box will come up with a number of options, which are saved with the project file, this dialog box is described below. For all subsequent times if it is required to set the project options then use the **Project | Set options for Project** menu option.

Whilst compiling the information window displays the file currently being compiled. Once compiled the message window will display the errors which were found in the program. Messages, and warnings are only displayed in the error file and the listing file (.ERR and .LST extensions). To immediately open the file and go to the line with an error then double click the mouse pointer on an error in the error window. Use the Alt F8 and Alt F7 keys to move forward and backwards through the list of errors and jump to their locations in the source files (The menu options **Compile | Next Error** & **Compile | Prev Error** perform the same operation).

Compiler Options Dialog Box

To show this dialog then use the **Project | Set options for Project** menu option The dialog box has a number of tabs each of which configures one area of the compilation.

Main



Processor

The processor which is to be used as a target is selected here. Note when a new processor is selected it will change the stack pointer, options and memory defaults. Using the “Show Only Devices Containing box” enter text to filter the list. For example type in “K50” in this box to show only processors which have K50 in their names. The Device Family box restricts to only those processors which are members of the 12/16 or 18 series families.

Click “New Processor” to select the processor and reset the other options.

Case Sensitive

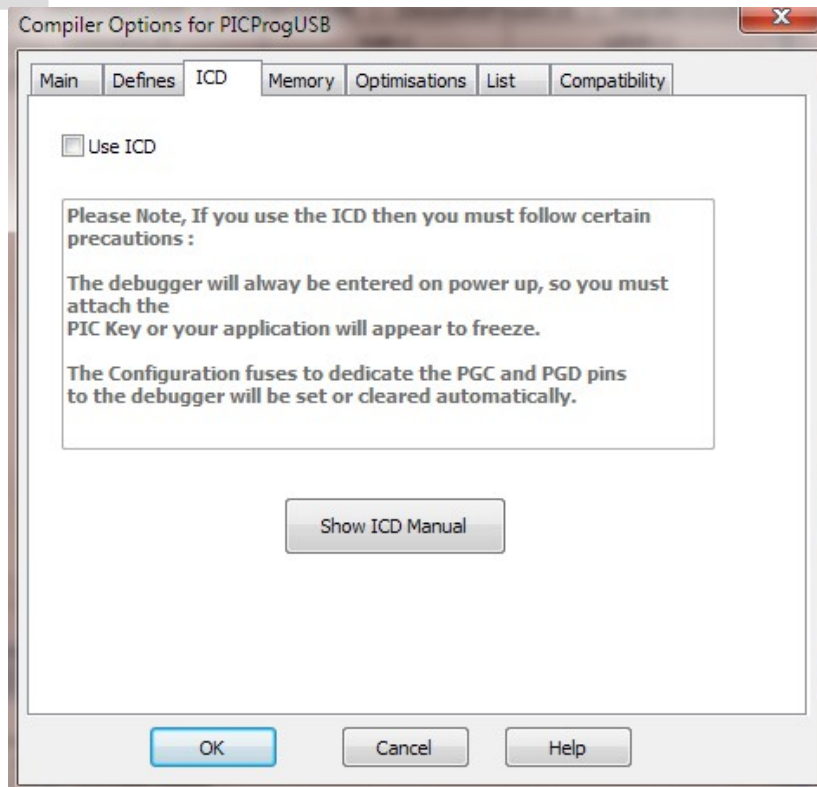
Normally this should be selected to ensure that the compilation is case sensitive.

Processor Frequency

The processor frequency is selected here, note that a constant integer called PROC_FREQ is defined and may be used in any of the C files in the project. The value defined is set to that of the Processor Frequency box..

ICD

The options available under this tab are described in the separate ICD manual:

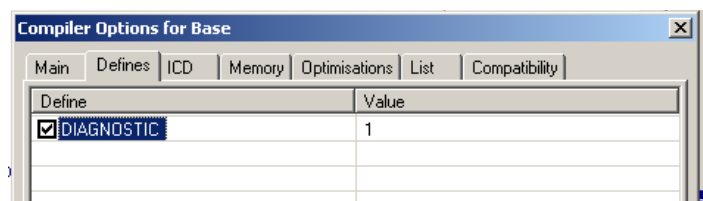


Note that the Use ICD box should be checked to include the ICD code, and that the manual can be shown using the button on this tab.

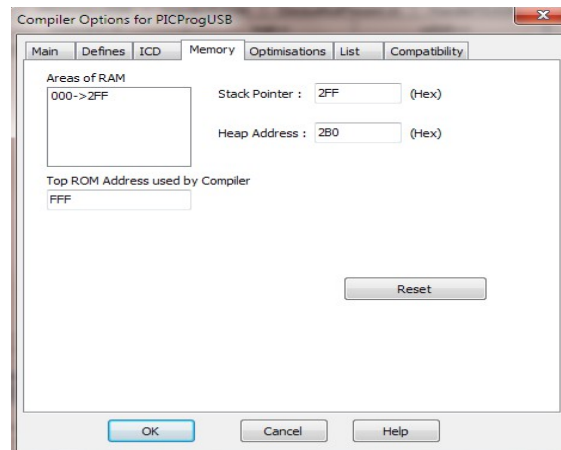
Defines

This box allows the user to enter a list of items which will appear as if the C pre-processor directive #define had been used.

For example if the label DEBUG is added with a value of 1, then for all files compiled the token DEBUG will be replaced with the value 1. The three buttons Add, Edit and Delete allow new items to be added, old ones to be deleted, or the value to be changed. The tick box allows the DEFINE to be included or not. If the tick is removed then the define will not be included. In the example below the value DIAGNOSTIC will be defined in each C file and replaced by the value 1.



Memory



Areas of RAM

This box shows the available areas of RAM for the selected processor, it may not be changed, but is shown for reference to assist with selection of the stack pointer.

Stack Pointer

The stack pointer is automatically assigned to the highest value of RAM when the processor is selected. It may be changed by the user to any value in memory to allow areas of memory to be set aside for program use, see the sections on Memory Allocation for further details.

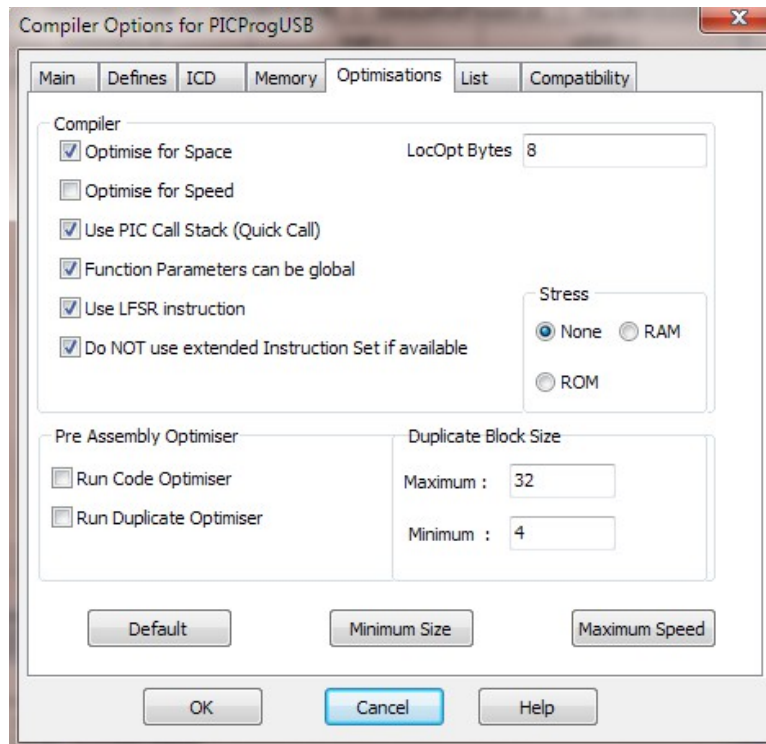
Heap Address

The Heap is used for storing interim results (e.g. when an array is returned from a function). It is automatically assigned to the lowest value of RAM in the highest memory page, or if there is only one RAM page then to the first free address in RAM. When there is more than one page it may be changed by the user to any value in memory to allow areas of memory to be set aside for program use, see the sections on Memory Allocation for further details.

Top ROM address used by compiler

This address allows the user to clear an area of ROM at the top of memory. Set the value to an address lower in memory to reserve free space for in-circuit debuggers, flash boot controllers etc.

Optimisations



Optimise for Space

This causes the compiler/assembler to generate code optimised for space. This option should normally be used. This option applies solely to the compiler, block optimisation is covered by the Pre Assembly optimiser (see below).

Optimise for Speed

This causes the compiler/assembler to generate code optimised for speed. This option applies solely to the compiler, block optimisation is covered by the Pre Assembly optimiser (see below).

Use PIC Call Stack (QuickCall)

This option causes the compiler to use normal PIC Call and Return instructions when selected, this allows up to the “depth of hardware” stack levels (between 29 and 124 levels for the 18X series) of function call within a C program (or one less if interrupts are in use). When not selected the software stack will be used which allows greater call depth, but is slower. Note that the main() function is not included in the call stack. With almost all newer PIC devices QuickCall is by far the most efficient and sensible approach.

Function Parameters may be global

This option allows the compiler to optimise function parameters to the LocOpt memory area instead of using the call stack. This is much faster than the call stack and very efficient. The compiler automatically determines when functions may use LocOpt rather than the stack. For example recursive functions (those which call themselves directly or indirectly) will never

use LocOpt. This option *should not* be used if your program uses pointers to functions - in this case the compiler cannot automatically determine how functions may call each other.

Local Optimise Bytes

This selects the number of bytes to be assigned in PIC global memory to which local variables and function parameters will be assigned if possible. Normally set to 16, can be increased if you have a large number of local variables in functions. This considerably speeds access to these variables.

Pre Assembly Optimiser

This optimiser runs after the compiler and before the assembler. It optimises in two ways. Firstly by searching for commonly repeated blocks of code which can be replaced by simpler constructs (Run Code Optimiser). Secondly by looking for duplicate blocks of code and replacing them by a sub routine which is called (Run Duplicate Optimiser). For the duplicate optimiser the Duplicate Block Size shows the minimum and maximum number of words in a block for which a replacement may be made. Typically this will be from 4 to 32 words.

Together these options typically save 10-25% from the overall code size although this is dependant on the source. For example a small program with use of char variables will probably not be optimised very much, a large program making use of large C types may show a much bigger reduction.

It is important that the duplicate block optimiser does not operate on assembler functions or files. To prevent this then use the following special key words :

Within C

```
#pragma optdup 0      // Turns off the optimiser for the rest of the file, or until
                      // the next optdup

#pragma optdup 1      // Turns on the optimiser for the rest of the file, or until
                      // the next optdup
```

Within Assembler

```
dupmodoff            ; Turns off the optimiser for the rest of the file, or until
                      ; the next optdup

dupmodon             ; Turns on the optimiser for the rest of the file, or until
                      ; the next optdup
```

List

These options are intended for compiler debugging and are of little use to the end users of the compiler. They are left available for end users to run specific tests under guidance of FED software engineers.

Compatibility

This tab provides a number of options for accepting non-standard syntax to allow files intended for 3rd party compilers to be compiled with the FED compiler. Please see [Compatibility Options](#).

On-line help

To obtain immediate help on any command or opcode then position the cursor in an editing window on to a keyword, and then

press Control and F1, if a help topic exists on that keyword then the help file will be opened at that topic.

Project Archiving

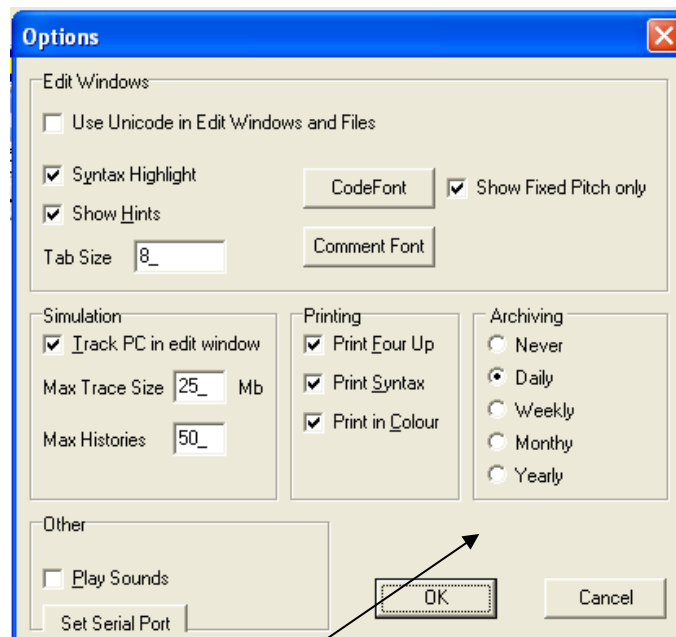
WIZ-C MX supports project archiving. This allows for all the files in a project to be stored together in a compressed archive. Archives can be created manually and will automatically include all project files and referenced header files, project definition and simulation files. However the most powerful use of archiving is the automatic archiving of project files at regular intervals. The interval can be set to daily, weekly, monthly or yearly. At the specified interval, and when the project is opened it will be archived. For example with a daily archive, then at the first time, on every day that the project is opened, then the source and project files will be saved in an archive file.

By default archives are created daily.

Archive files are small, a typical medium sized project will have archives of around 25K in size. Archive files are saved in the Archive folder under the main project.

Automatic Archiving Interval

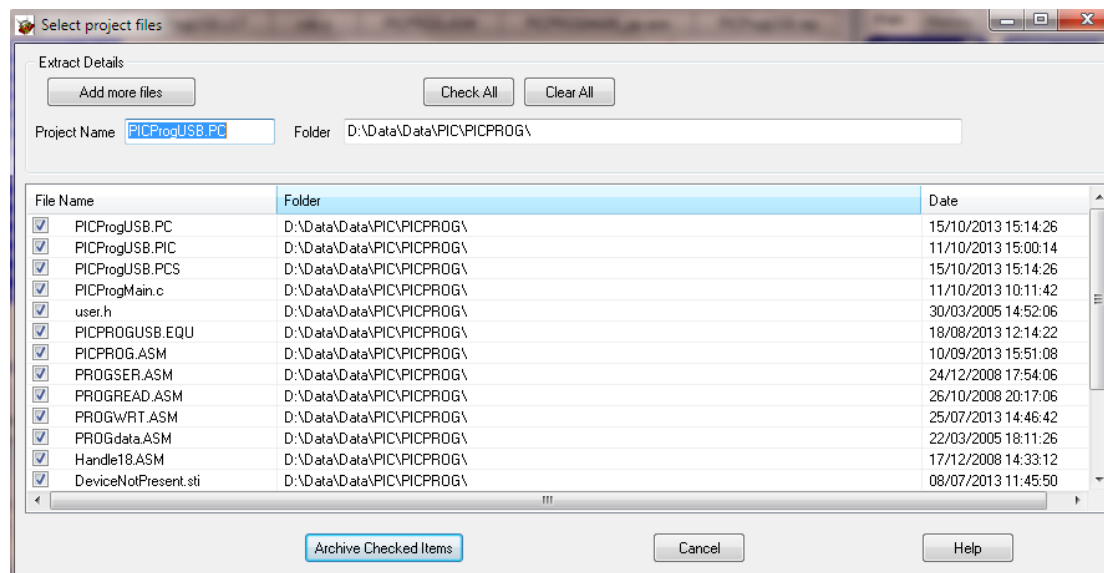
To set the interval at which archive files are created then use the **File | Options** menu :



The interval may set in the archiving box.

Manual Creation of Archives

Archives may be created manually. To do this use the **Project | Archive** Project menu option. This will firstly allow a file name for the archive to be selected, and then bring up the archive dialog box :

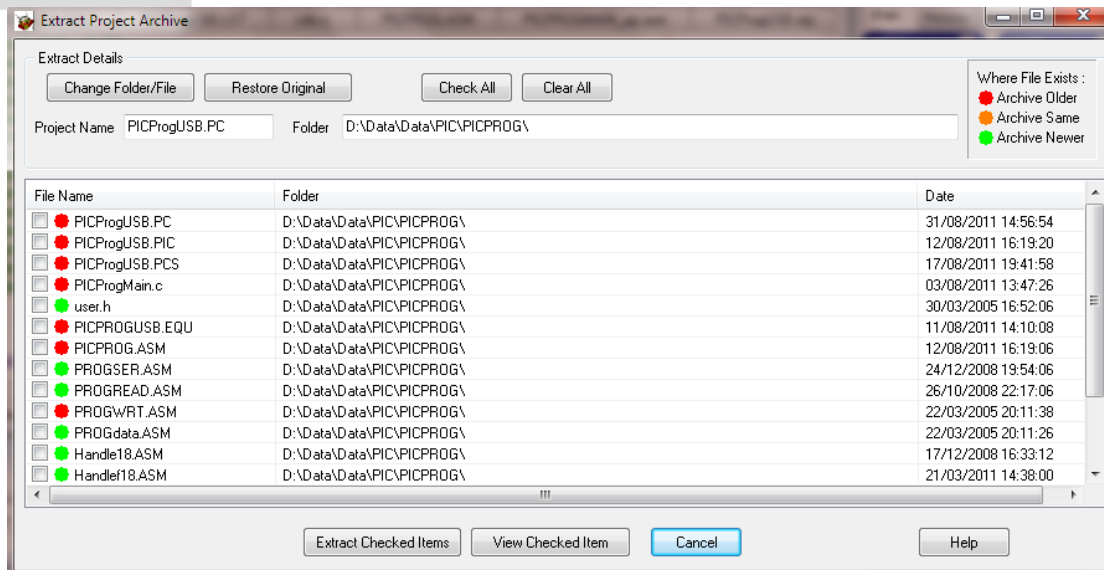


By default all the files in the project, the WIZ-C definition files which support the project, and any headers found in source files will be included. The archive includes the folder from which the files were originally saved. It is possible to include additional files here – the Add More Files button allows other files to be selected. Please note that in the automatic archiving any additional files are not included, the best way to ensure additional files are included in automatic archives is to include them as comment files in the project window.

Click the “Archive Checked Items” button to copy the selected files to the archive.

Restoring Archives

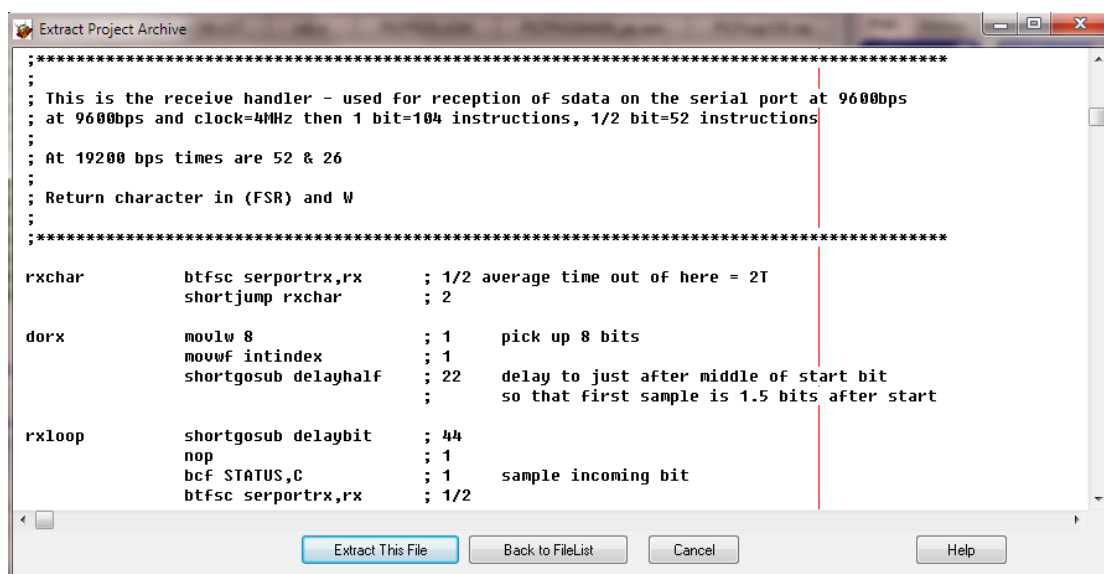
To restore archive files then use the Project | Restore Archive option, this will allow an archive file to be selected (archive files are saved in the Archive folder under the project). Then the dialog box to allow files to be extracted will be shown :



This shows the list of files in the archive and the folder to which they will be extracted. To change the folder then use the Change Folder/File button. This will allow the folder and/or project file name to be changed. Note that if the project file name is changed then all the files which have that name as a base will be changed as well, however external files, or header files will not be changed.

The traffic light shown by the file name is only shown when the file exists and is a warning to the user that the file exists. It is red if the archive file is older than the current existing file, amber if the archive and existing file is the same date, and green if the archive is a newer date than the existing file. It is up to the user to then decide which files to extract by checking the box next to all those files. The check all and Clear All buttons act to check or clear all tick boxes at once.

To view a file click the "View Checked Item" button, this will open the first selected file for examination :



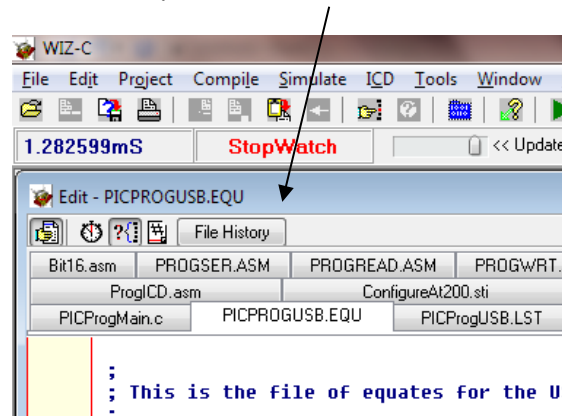
Here you can read the file and scroll up and down, select text and copy to the clipboard (using Ctrl+C or the right menu key), or extract just this file using the button. Return to the

file list using the Back to File List button, or cancel out if you just want to copy text to the clipboard and paste into an existing file.

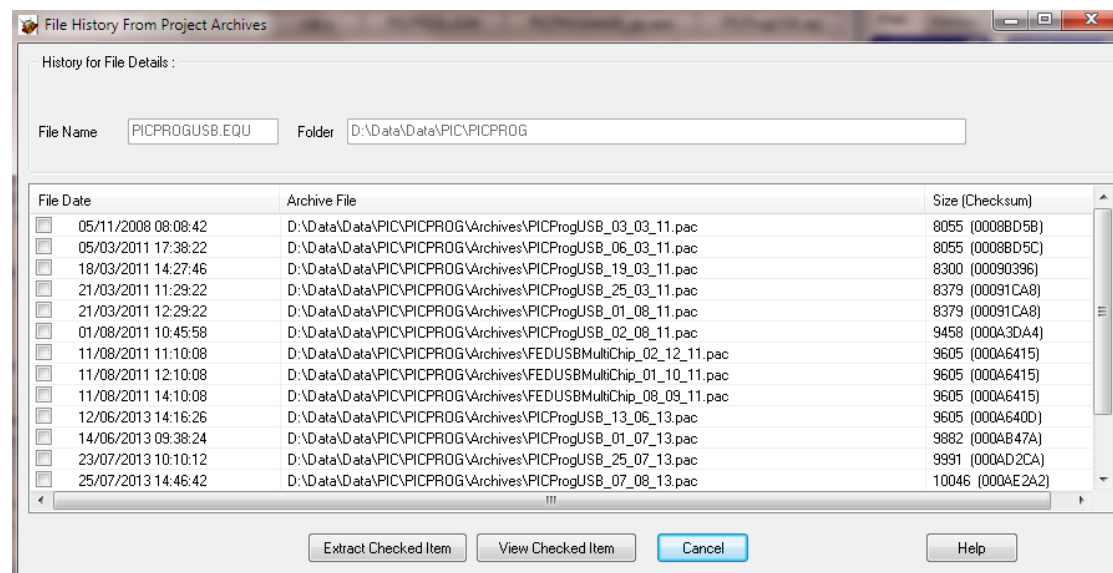
Finally files will be extracted if the Extract Checked Item button is used.

File History

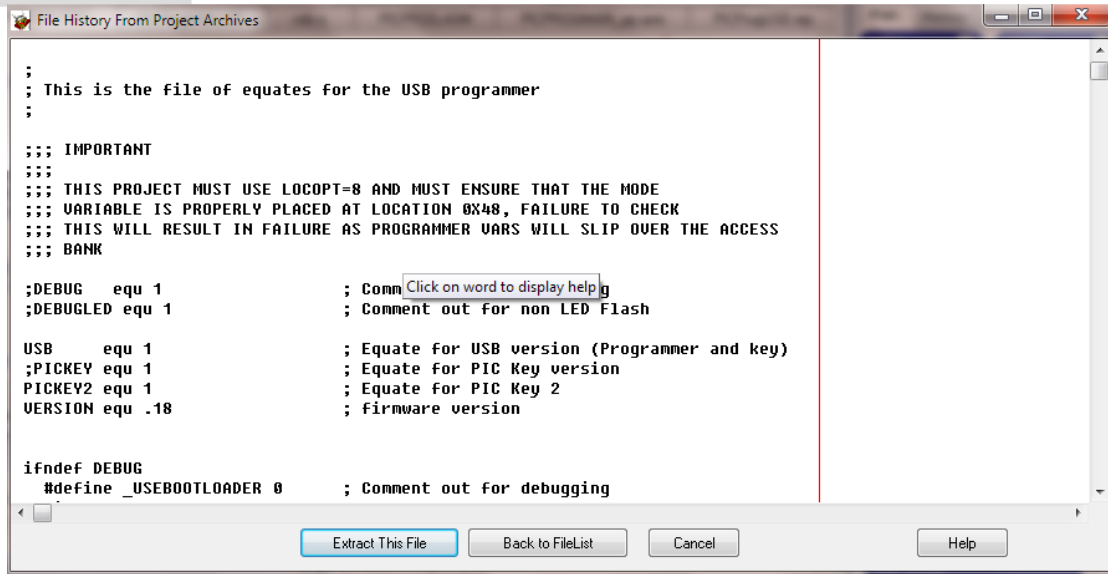
Rather than examining all the files in an archive it is possible to examine all the versions of a file held in every archive. To do this Open a file which you wish to examine in the editor, now use the File History button at the top of the edit window :



In this example we want to see all the previous versions of the file "PICPROGUSB.EQU" held in the archives folder, so the File History button is pressed. This brings up the File History list :



Here you can see that file history for this file goes right back to 2008. Click any file to either extract it, or view that file from the archive using the View Checked Item button :



Here you can read the file and scroll up and down, select text and copy to the clipboard (using Ctrl+C or the right menu key), or extract just this file using the button. Return to the file list using the Back to File List button, or cancel out if you just want to copy text to the clipboard and paste into an existing file.

Finally files will be extracted if the Extract Checked Item button is used.

Menu Commands

The following sub-menus are available:

- File Menu**
- Edit Menu**
- Project Menu**
- Compile Menu**
- Simulate Menu**
- ICD**
- Tools Menu**
- Window Menu**
- Help Menu**

File Menu

The File menu contains the following commands:

New : This creates a new empty file ready for entering text. To give the new file a name then save it, this will prompt for a file name.

Open : This brings up a dialog box which allows an existing file to be opened.

Reload All Files: This reopens all the currently open files.

Libraries : This allows additional libraries to be added to the standard list. See the chapter on Creating Libraries for more details - Set Libraries Dialog Box

Save : This saves the Current File.

Save As : This saves the Current File allowing a new file name to be chosen.

Save All : This saves all open files which have changed.

Insert File : This brings up a dialog box which allows an existing file to be inserted in to the current file at the current editing location.

Close Page: This closes the file which is currently displayed in the edit window.

Close All Pages: This closes all open files which are currently displayed in the edit window.

Print : This prints the Current File

Print All : This prints all open files.

Printer Setup : This brings up the printer setup dialog.

Options : This brings up a dialog box which allows various options to be set including the number of pages printed on each printer page. It allows the user to set the tab size, turn Syntax Highlighting and Hover Help on and off and allows the maximum size of the simulation trace file to be set. In addition code and comment fonts may be set and sounds may be turned on and off.

Past Files: At the bottom of the menu is shown a list of recent fiels, click any of these to open that file.

Exit : This leaves the program, prompting for any changed files to be saved, and saves all the current Project information in the project file.

Edit Menu

The Edit menu contains commands which are concerned with the Current File and are as follows:

Cut (Delete) (or Ctrl+X): This command deletes the text currently selected and copies it to the clipboard

Copy (Ctrl + Insert) (or Ctrl+C):This command copies the currently selected text to the clipboard where it may be pasted in to any open file in BASIC, or in any other file.

Paste (Shift + Insert) (or Ctrl+V): This command pastes the text currently in the clipboard into the current editing window at the location of the cursor.

Undo (Alt + Backspace) : This command undoes the most recent typing, or the most recent delete command.

Clear : This command clears the entire editing file. If the file has been changed since it was last changed then the user will be prompted to save the file before it is cleared.

Select All : This command selects all text in the current file.

Copy Device Picture to Clipboard: This command copies the PIC outline and named pins to the clip board in windows picture format suitable for pasting into another application – only for WIZ-C users.

Find/Replace (Shift + F3) : This command brings up the Find/Replace dialog box which allows text to be found in the file being currently edited.

Repeat Find/Replace (F3) : This command repeats the last find or replace operation. If the last operation was on a different file then the text used for the find or replace operation on the different file is still used for the operation on the current file.

Set Bookmark (Shift+Control+n): This commands set a mark in the file. To return to the same point later then use Jump Bookmark n. Note that up to 10 bookmarks can be set, use control+shift+number keys 1 to 9 to set a different bookmark number.

Jump Bookmark (Control+n): This command jumps to the previously marked point in the file. Note that up to 10 bookmarks can be set, use control+number keys 1 to 9 to jump to a different bookmark number.

Clear All Bookmarks : This command clears all bookmarks from the file.

Goto Line: This command brings up a dialog box allowing any specific line number to be brought up.

Goto Label: This command brings up a dialog box allowing the line containing the entered label in the assembled program to be shown in the edit window.

Goto Label Under Cursor: This command shows the line (containing the label under the cursor) in the assembled program in the edit window.

Project Menu

The Project menu contains a series of commands which are associated with the management of the Project. Note that some of these commands change when The Professional Version is in use, the commands are the same as below but refer to project groups or the current project. The commands are as follows:

Open/New Project : This commands opens an existing project file, or creates a new project file. A project file contains a list of all the files which make up the project, and saves a list of all the files which are open for editing. The project file also saves the options selected for the project. When a project is opened then any current project is automatically saved first. To create a new project then enter a project file name which does not yet exist. The project will be created with this name automatically.

Close Project : This command closes the current project and opens a new blank project.

Archive Project : This command brings up the Project Archiving system for archiving the project.

Restore Project : This command brings up the Project Archiving system for restoring the project from an archive.

Current Project Options : This command brings up the Compiler Options Dialog Box.

Save Project As : This command brings up a dialog box and allows a new project and path name to be entered. Note that all files are saved relative to the new path, so that the project window files are not copied - they simply show a link to the position of the file in the old project.

Add Project to Group : Professional users only – this will add a new project to the project group in a multi-project simulation..

Delete Project from Group : Professional users only – this will delete a project from the project group.

New Projects use settings from Current Project : This forces any new projects created to use the same processor, clock rate, window layout etc. as the current project.

Add/Insert Item (Insert key when the Project Window is active) : This command adds a file name to the Project Window, or inserts before the currently selected file.

Delete Item (Delete key when a file is highlighted in the Project Window) : This command deletes the file in the project window which is currently highlighted from the project.

Modify Item : This command allows the type of the currently selected item to be changed, for example changing a C file to a text comment file..

Show as Icons: This command shows items in the project window as icons instead of a list when selected..

Use Application Designer : For WIZ-C users this menu option turns the application designer on or off. If turned off then the system works identically to the FED PIC C Compiler.

App Designer Verify : For WIZ-C users this menu option verifies that all elements included in the design are correctly connected..

Open All Files: This command opens all the files in the project.

Past Projects: At the bottom of the menu is shown a list of recent projects, click any of these to open that project.

Compile Menu

The Compile menu brings up a list of commands which are applicable to compiling the entire Project . The menu contains the following commands.

Generate Application (Ctrl+F9) : This command for compiles all projects in the project group.

Compile Current Project (Alt + F9) : This command compiles all the C files which are members of the current Project only. This command brings up the Compiler Options Dialog Box.

Next Error (Alt + F8) : Once the project has been compiled, there may be errors. The errors are listed in the Message Window This command will find the next error in the error list and bring up the file with the error with the cursor at the line which contains the error.

Prev Error (Alt + F7) : This command operates in the same way as the Next Error command, but finds the previous error.

Simulate Menu

Please see the simulator help file (under **Help | Simulator contents**) for more details on this menu which is identical to the PICDESIM Simulator menu and is described in the PICDESIM simulator manuals.

ICD

This menu contains items of specific use for the In Circuit Debugger which is explained within its own manual.

Tools Menu

The tools menu includes the Run MPLAB (C) and Run MPLAB direct options. Please see: [Use with MPLAB.](#)

The Tools menu contains a number of standard user defined commands which may be added at will. The user defined commands may be changed or deleted as required by using the **Tools | Configure** menu option

Configure : This command allows the list of tools to be changed. It brings up the [Tools Configuration Dialog](#) box.

Tools Configuration Dialog

This dialog box lists all the current tools. To remove a tool from the tools menu then select a tool in the list and click the remove box. To add a new tool then decide on a name for the tool and then enter it into the edit box in the form of the name, then a comma, then the full path of the command. Put an & character in front of the character to be used for the shortcut key. For example:

```
File &Manager,C:\windows\winfile.exe
```

To add command line arguments then special characters %f and %p may be used. %p inserts the current project name WITHOUT an extension. %f inserts the current editing filename for the edit window which is currently being used, with its extension. %t inserts the name of the current processor. If a file replacement (%f or %p) is followed by an underscore (" _") then the short form of the filename is used for older applications. For example this is the line for the FED PIC Programmer:

```
Pic &Programmer,C:\Program Files\FED\PICPROG.EXE %p.HEX /P%t
```

When the OK button is clicked then the menu is changed for the new tools. The tool list is not stored with the project file, but is stored in the global initialisation file.

The maximum number of tools is 10.

Window Menu

This menu contains the following commands:

Small Fonts: This is a toggle option which reduces the size of fonts in the Project, Information, and Debugging Windows, allowing more information to be shown on lower resolution screens.

Arrange for Debug : (ALT+D) This command arranges all the windows on the main FED PIC C window so that they are visible and sized according to their function particularly for simulation and debugging.

Arrange for Edit : (ALT+E) This command arranges all the windows on the main FED PIC C window so that they are visible and sized according to their function particularly for when a project is being edited.

Arrange in compact form: (ALT+C) This command arranges all the windows on the main FED PIC C window so that they are visible and sized with a smaller debug window.

Tile : This command tiles all the sub windows on the main screen, so that they are all visible simultaneously.

Help Menu

This menu contains the following commands:

C Compiler Contents (F1 key) : This command brings up the contents list of the help file.

Assembler Help : This command brings up the contents list of the help file for the assembler which is identical to the PICDESIM assembler.

Simulator Contents: This command brings up the contents list of the help file for the simulator which is identical to the PICDESIM simulator.

Elements: A help file for WIZ-C - showing the WIZ-C element reference.

Application Designer: A help file for the WIZ-C application designer.

MPLAB: A help file showing how MPLAB should be used with PIC C Compiler and PICDESIM.

Lookup Keyword (Ctrl+F1) : With the cursor positioned on a key word in the Current File then this command will bring up the help file with the topic describing that key word.

About : Shows version number and copyright information about FED PIC C..

Windows

All the Windows in FED PIC C contain a "speed menu", use the right mouse key when the pointer is over the window to bring up a list of menu options relevant to that window. The following windows are open permanently:

Edit Window

Project Window

Information and Error Window

Debugging Window

Edit Window

The edit window holds all the current files which are being edited. Each file is shown on a tab. Click a tab to bring that file to the front.

To open a file click the file in the Project Window, or use the **File | Open** menu option. To open all the files in the project use the Project Window menu (right click the window and select the **Open All Files** menu option). To open an include file then position the cursor on the line with the #include statement and press Ctrl+Enter.

To close a file then use the **File | Close Page** menu option (or press Ctrl+L). To close all the pages use the **File | Close All Pages** menu option.

There are a number of Edit Window related menu options which are duplicated on the right mouse button in the Edit Window.

Project Window

A window which holds a list of files to be compiled. Each type of file has its own tab in the project window.

The project window shows a list of all the files which are associated with the current program. When files are added they are selected as being of type C, H, Assembler, Stimulus/Injection or Comment.

C and H type files are compiled. Stimulus/Injection files are used by the simulator. Comment type files are ignored by the compiler, and are used normally for documentation or instructions for use with the project.

Double click on a file in the project window to open that file for editing. Select a file and press Enter to change the file type. To add and delete files use the commands on the Project Menu, or right click the window. Files are added to the project window before the selected file (if one is selected), or at the end of the list if no files are selected.

There are a number of Project Window related menu options which are duplicated on the right mouse button in the Project Window.

Information and Error Window

The information window contains information about the current compilation. Information is included about the files being compiled, copyright statement, and the number of bytes occupied by the program.

The Error Window in the lower half of the Information window shows the current list of errors. See [Compiling a project and reviewing errors](#)

Debugging Window

The debugging window is used in the same way as the PICDESIM Simulator, please see the help file for PICDESIM for more details.

In [The Professional Version](#) there is one debugging window for each project which is a member of the project group.

6 *Example projects*

The FED Development Board

Using an LCD Display

EEPROM Programmer

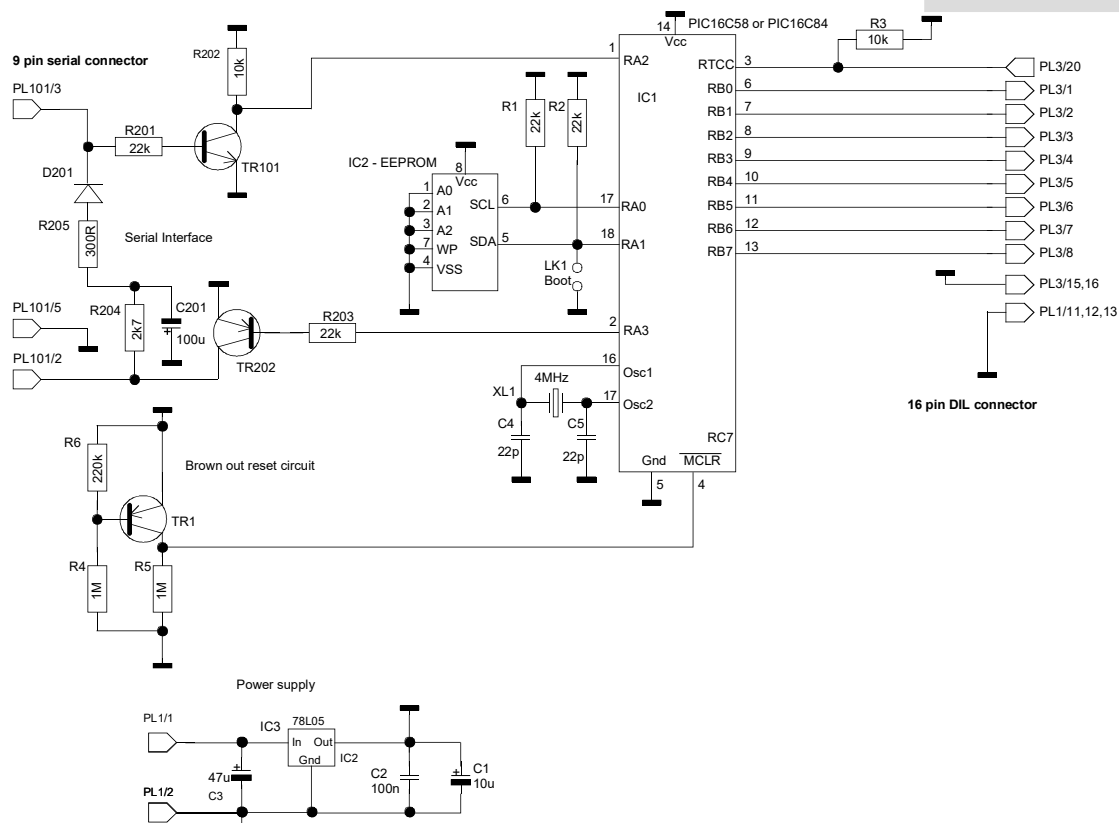
Example projects are shown here for all versions – see The Professional Version for examples which apply to that version only.

The FED 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. (Version 9 note – please note this board is no longer available). The board has the following features:

- Supports all 18 pin PIC's
- Has on board 5V regulator - supports 100mA or 1A regulators
- Serial interface compatible with PC/MAC serial interfaces.
- Supports 8 pin EEPROM/RAM devices with IIC interface
- 4/20MHz crystal oscillator
- Port B available on 16 pin DIL connector

The circuit diagram is shown below:



IC3 provides the power supply, the 78L05 may be used for lower consumption projects, the 7805 can be used for projects of up to 1A consumption. In target projects IC3 may be left out and replaced by the power supply used in the main circuitry.

I2C is the EEPROM or RAM. Devices of any size may be used provided that they have the pin out shown in the circuit diagram, and that they have the I²C interface. Two types of EEPROM are common, 12 bit address devices (e.g. 2Kbyte and below), and 16 bit address devices (e.g. 8Kbyte). R1 and R2 pull up the data and clock lines used on the I2C interface and should be 2K Ω with a 20MHz clock.

IC1 is the PIC. The lower 4 bits of port A are used for the EEPROM interface, and also for the external serial interface, these bits are not available for use by the application, however the RTCC input (which forms the RA4 input of the PIC16C71, and PIC16C84) may be freely used.

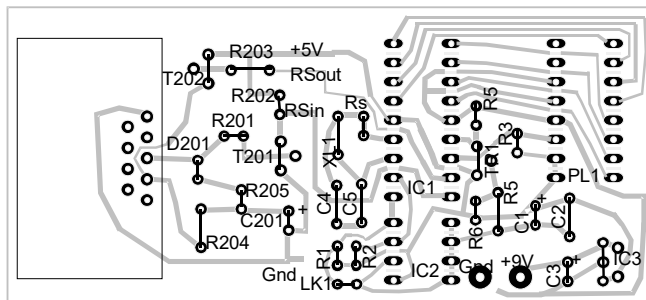
The oscillator circuitry around pins 15 and 16 of the PIC is based on crystal or ceramic resonator devices. The frequency of the oscillator may be 4MHz or 20MHz

The serial interface is provided as standard, the FED PIC C `serin()` and `serout()` functions may be used to drive the interface. The interface at the PIC is inactive high - i.e. when the interface is idle the input and output of the PIC is held at +5V. The start bit is signalled by a low going pulse followed by the 8 data bits. C201 stores the negative voltage which is normally present on the inactive input line, filters it, and uses it as a negative supply for the signal driven from the module. This circuit may only be used for communication with other systems which provide a standard RS232 output, and worked with all PC's with which it was tested.

The Component list is shown below:

Resistors		Semiconductors	
R1,2	22K	IC1	PIC - see text
R3	10K	IC2	24LC16 etc. see text
R4,R5	1M	IC3	7805 or 78L05
R6	220K	TR201	BC548
R201	22K	TR1,TR202	BC557
R202	4K7	D201	1N4148
R203	10K		
R204	2K7	Other	
R205	300R	XL1	4.000MHz crystal or ceramic resonator
Capacitors		PCB	
C1	10uF 10V Electrolytic	PL101	9 pin D socket
C2	100n, Ceramic	PL1	16 pin DIL IC socket
C3	100uF 10V Electrolytic	LK1	0.1" link with jumper
C4,5	15pF, Ceramic	IC sockets	8pin, 18pin
C201	100uF 16V Electrolytic	Veropins	2
		Heatsink	IC3, optional

The PCB layout is shown below:



See the Example Program using the demo board

Example program using the demo board

The program below (in the "Serial Test" directory, project "Dump") is a very simple test of the demonstration board. When the call dump() is executed all PIC memory variables from Start to End are sent in hex form to the attached PC. This is a useful diagnostic to show values of variables as a program is running.

The demonstration program dumps all variables from 5 to 20 hex. The information is shown in the form:

```
05=1F
06=87
07=00
etc
```

The project can be opened and the program compiled and assembled in PICDE. It can be programmed to a PIC16F84. The system uses a 4MHz clock and no watchdog timer. Use the terminal on PICDE to display information from the board. When power is applied the memory dump will be shown on the terminal

```
//
// Simple diagnostic, write all variables from start address to
end address
// to attached PC
//

#include <pic.h>
#include <DataLib.h>

void Dump(BYTE Start,BYTE End);
void PrtHex(BYTE n,BYTE *s);

void main()
{
    PORTA=0x1F;
    TRISA=0x17;
    Dump(0x5,0x20);
endit:
    while(1);
}

//
// Dump in hex to serial port on PORT A, bit 3
//

const int SERIAL_RATE=9600;           // Set serial port rate
const int BITTIME_IN=PROCFREQ*1000/SERIAL_RATE/4;
const int BITTIME_OUT=PROCFREQ*1000/SERIAL_RATE/4;

const int SERIALPORT_IN=5;           // Port for serial i/f
const int SERIALBIT_IN=2;            // Bit for serial i/f

const int SERIALPORT_OUT=5;          // Port for serial i/f
const int SERIALBIT_OUT=3;           // Bit for serial i/f

void Dump(BYTE Start,BYTE End)
{
    BYTE i;
    BYTE d[2];

    for(i=Start; i<=End; i++)
    {
        BYTE v=(BYTE *)i;
```

```

    PrtHex(i,d); pSerialOut(d[0]); pSerialOut(d[1]);
    pSerialOut('=');
    PrtHex(v,d); pSerialOut(d[0]); pSerialOut(d[1]);
    pSerialOut('\n');
  }
}

//
// Print Number in hex to 2 character string
//
void PrtHex(BYTE n,BYTE *s)
{
  BYTE l,h;

  l=n&0xf; h=n>>4;

  s[1]=l+'0'; if (l>=0xa) s[1]+=7;
  s[0]=h+'0'; if (h>=0xa) s[0]+=7;
}

```

Using an LCD display

Introduction to LCD Displays

Complete LCD Example

Introduction to LCD Displays

Functions are provided 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 port B, there is no option to change the default port. The pin connections are as follows:

LCD Module	LCD Port number	Pin Number (2 line display LM016L)
RS	B1	4
R/W	B2	5
E	B3	6
D4	B4	11
D5	B5	12
D6	B6	13
D7	B7	14
Vss	-	1
Vdd	-	2
Vo (LCD Supply)	-	3

Connections D0, D1, D2 and D3 on the module can be left floating.

The LCD display has an 8 bit interface, but read and write operations are executed in two 4 bit transfers. For ports which do not have pull up resistors the D4 to D7 signals should be pulled up to +5V with 10k-100K resistors. Bits 0 to 3 of the LCD should be left floating, or tied high. Port bit 0 on the LCD port is still available for general use, in this case the tris-state command for the LCD port should be set to drive on bits 2 , 3 and 4, and to read on bits 4, 5, 6, and 7. Bits D4 to D7 are available for general purpose inputs when the module is not being used, but in this case should be coupled with resistors to allow the main program to overdrive the inputs when the module is being written.

The functions used to drive the module are LCD and LCDString. The LCD function takes one parameter. This has different values depending on the function as described below. The LCDString function writes a complete string to the display.

The program must define the port to be used with the display. This is done by defining a constant integer which must be set to the port. For example to use port B then the line shown below is used:

```
const int LCDPORT=&PORTB;
```

It is possible to connect the E, RS and RW pins of the LCD display to other pins of the PIC - although the data pins must still be connected to bits 4 to 7 of the LCDPORT. For example here is how to define an LCD where the data bits (D4-D7) are connected to port D bits 4 to 7, E is connected to PORTE bit 1, RS to port D bit 2, and RW to Port D bit 3. NOTE YOU ONLY NEED TO DO this if the LCD is connected other than as shown above.

```
const int LCDPORT=&PORTD;
const int LCDEPORT=&PORTE;
const int LCDEBIT=1;
const int LCDRSPORT=&PORTD;
const int LCDRSBIT=2;
const int LCDRWPORT=&PORTD;
const int LCDRWBIT=3;
```

To initialise the display which must be done before any information is written, the LCD function is used with a negative number. To initialise a one line display then use the function LCD(-1), alternatively to initialise a two line display then use the function LCD(-2). The LCD function defines the Port to drive on the correct bits, the display is set to the specified number of lines, the display is turned on, and the cursor is set to an underline at the start of line 1.

To write a character to the display then use the LCD() function. Thus to write an A character to the display use LCD('A').

To send a function to the display then add 256 to the function number and use the LCD function. These functions are documented in the Hitachi controller driver documentation, however a summary of some of the more important functions is included below:

Function

Function

LCD(-1);	Initialise display to 1 line
LCD(-2);	Initialise display to 2 lines
LCD(257);	Clear display, return cursor to home position
LCD(258);	Return cursor to home position
LCD(256+128+N);	Return cursor to line 1, position N, where N=0 is the first character on line 1
LCD(256+192+N);	Return cursor to line 2, position N, where N=0 is the first character on line 2

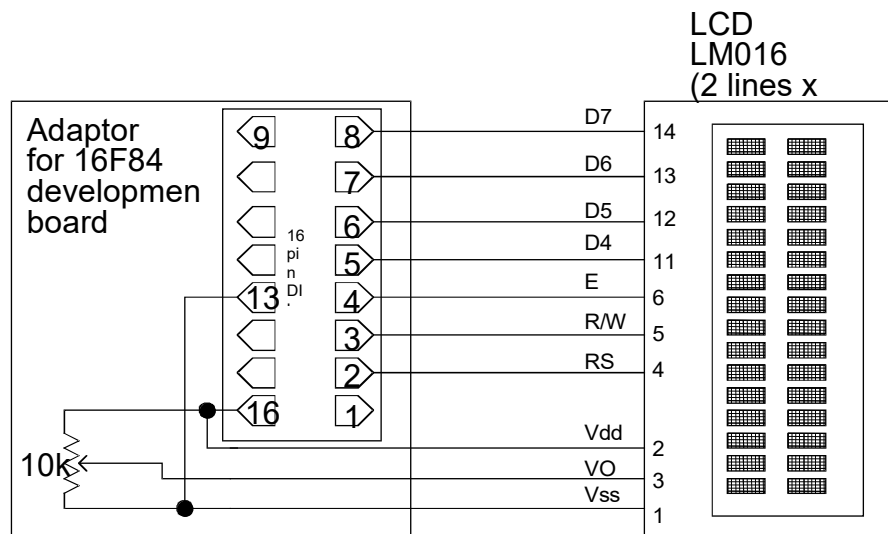
The LCDSTRING function sends the supplied string to the display. Thus to write "HELLO" to the display then the following can be used:

```
LCDSTRING("HELLO")
```

HINT

Many single line displays are actually implemented as two 8 character lines which are connected to appear as one 16 character line. Use LCD(-2) to initialise these displays, and print on the second line to print to the second 8 characters.

Complete LCD Example



The board with the 16F84 is used for the complete example, an interface board with a 16 pin socket should be wired as shown below:

The example program below (which is in the LCD directory) shows a typical interface to this circuit. Note that this program is not very efficient on space owing to the extensive use of integer types.

Version 9 update – The copy in the LCD directory (which is sub-directory of the projects directory) is now modified to operate on the FED Development board with a 16F877 and includes an LCD device simulation. It needs to be run for several simulation seconds to see the display updating.

```

#include <displays.h>
#include <delays.h>
#include <pic.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=6;          // Define LCD port as Port B

void main();
void LCDClear();             // Clear LCD display and set cursor to
origin
void LCDNum(int number);      // Print a number to the display

void main()
{
    int i;

    LCD(-2);                 // 2 line display

    LCDClear();
    Wait(1000);
    LCDString("LCD Display");
    Wait(3000);
    LCDClear();
    LCDPrintAt(0,0);
    LCDString("At 0,0");
    LCDPrintAt(1,1);
    LCDString("At 1,1");
    Wait(3000);
    LCDClear();
    LCDOnOff(1,1,1);
    LCDString("Blinking");
    Wait(3000);
    LCDClear();
    LCDString("Shift left");
    LCDPrintAt(0,1);
    LCDString("<<<<<<<<<");
    Wait(1000);
    for(i=15; i; i--)
    {
        Wait(200);
        LCDShift(0,0);
    }
    LCDClear();
    for(i=-5; i<=5; i++)
    {
        LCDNum(i);
        Wait(512);
        LCDPrintAt(0,0);
    }
    LCDClear();
    for(i=1; i<10000; i+=i)
    {
        LCDNum(i);
        Wait(512);
        LCDPrintAt(0,0);
    }
    Wait(2000);
    LCDClear();
    LCDString("Tests Complete");
    while(1);
}

void LCDClear()
{

```

```

LCD(0x108);    // Display off
LCD(0x101);    // Clear display
LCD(0x10C);    // Display on
LCD(0x180);    // Print at 0,0
}

void LCDNum(int number)
{
  BYTE numflag=0;
  if (number<0) {LCD('-'); number=-number;}

  if (number>9999)
    {LCD(number/10000+'0'); number=number%10000; numflag=1;}
  if ((number>999) || numflag)
    {LCD(number/1000+'0'); number=number%1000; numflag=1;}
  if ((number>99) || numflag)
    {LCD(number/100+'0'); number=number%100; numflag=1;}
  if (number>9)
    {LCD(number/10+'0'); number=number%10;}
  LCD(number+'0');
}

```

EEPROM Programmer

Programmer Introduction

Driving the 24LC65

Programmer

C Program for the PIC

PC Application Program

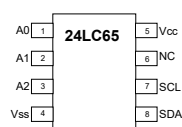
Further notes on using I2C EEPROMs

Programmer Introduction

Microchip (and others) manufacture I²C EEPROM's in a range of sizes. In this example we shall construct a programmer for the 24LC65 (and compatible devices), which is an 8Kx8 memory and which is suitable for data logging applications. For simple storage of configuration variables smaller devices are probably more suitable, however all the devices are driven in similar fashion.

Driving the 24LC65

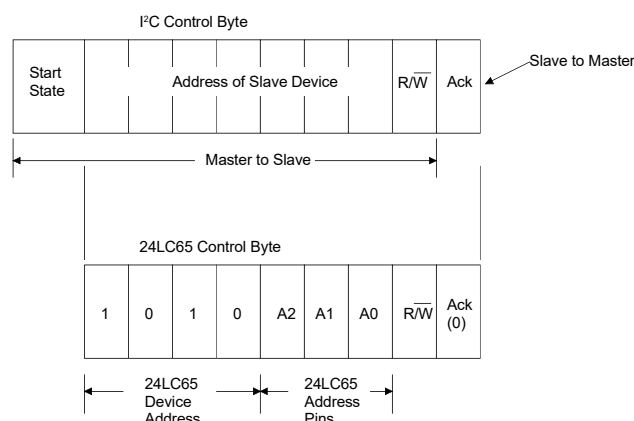
The figure below shows the pinout of the EEPROM. The SCL and SDA pins are for the I²C bus, the address pins A0, A1, and A2 are connected to ground or Vdd. When the device is addressed these bits are used as part of the I²C address selector to choose the particular



device. By using these pins it is possible to use up to 8 devices on a single I²C bus, therefore allowing up to 64K of EEPROM to be addressed on a 2 wire bus.

The first byte transferred on the I²C bus is the control byte which selects the specific chip to be addressed. The figure shows the control byte for the 24LC65, note that the bottom 3 bits of the address within the control byte for the 24LC65 are the address bits which match the pins on the device.

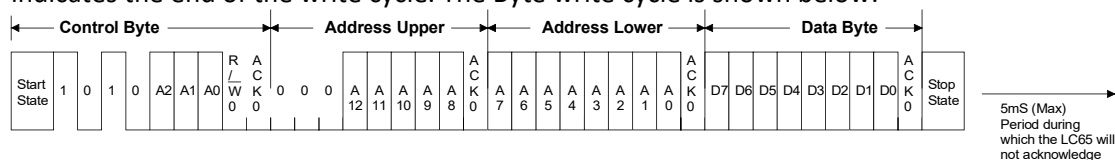
The 24LC65 has a number of modes to access the memory array. We will not go into all of



these modes, but will simply consider the Byte Write, Random Read, and Current Address (sequential) Read modes.

Byte Write.

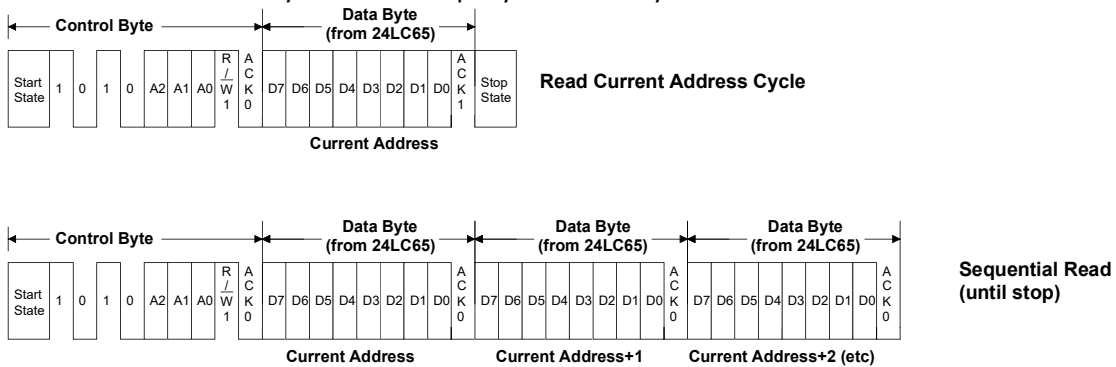
This mode allows a single byte of data to be written to any address in the memory. The control byte is written first with the R/W bit set to 0 to indicate a write, this is followed by the address which is transmitted as two byte transfers to the EEPROM, the most significant byte of the address is written first, followed by the least significant byte. As in all other transfers the data is written Most Significant Bit first. The final byte written is the data byte. At the end of each of the 4 bytes written the 24LC65 generates an acknowledge bit. After all 4 bytes have been written the master device generates a stop state and the 24LC65 initiates an internal write cycle. The time taken to write the data is guaranteed not to exceed 5mS, so the master may either wait for this time to allow the data write to complete, or it may continuously poll the device sending control bytes until the 24LC65 acknowledges which indicates the end of the write cycle. The Byte write cycle is shown below:



Read current address - sequential reads.

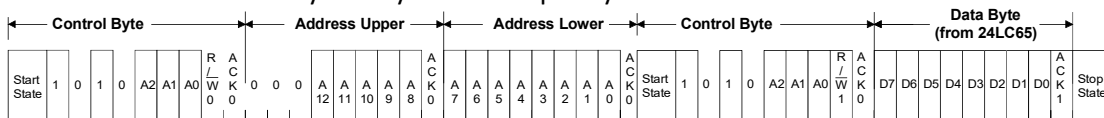
To read a byte from the EEPROM then the same control byte is used but with the R/W bit set to 1. Following the acknowledge bit from the 24LC65, the master sends an additional 8 clocks to the 24LC65, the 24LC65 then clocks out the requested byte. Following the last bit sent by the 24LC65, the master may either send an acknowledge bit, or will not acknowledge. If the master acknowledges then the 24LC65 increments the current address,

and the master may then send a further 8 clocks to read the next byte, this allows the entire contents of the memory to be read rapidly. The read cycle is illustrated below:



Random Read.

A random read is performed by setting the current address as if a write cycle were to be undertaken. However following the lower byte of the address another start condition is generated by the master, this is followed by a control byte to read the current address, and then the contents of the supplied address may be read. The figure shows the random read cycle. Note that the random read cycle takes 5 byte transfers on the I²C bus, and using the routines shown last month this can be as much as 500uS. Therefore, wherever possible the random read should only be used when absolutely necessary, use sequential reads otherwise which take only one byte transfer per byte read.



Other modes.

The other write mode which is also very useful is the page write mode which allows up to 64 bytes to be written simultaneously and therefore can increase write speed by up to 64 times. In similar fashion to the sequential read the master continues to send up to 64 data bytes before the stop state, and when the stop is received all 64 data bytes are written to the memory simultaneously. This mode is not shown further as the code required to drive it would occupy too much space.

It is possible to program internal security bits to prevent further writes to the EEPROM array.

The 24LC65 device has two EEPROM areas internally, one has the capability for considerably more write operations than the other, this is known as the high endurance memory. There is also a mode available to re-map the high endurance memory internally.

Programmer

The EEPROM programmer is based on [The FED development board](#). The 24LC65 fits into the 8 pin IIC socket - IC2. This socket is also pin compatible with other EEPROM devices such as the 24LC16. The programmer communicates with a PC over the serial port on the board.

The programmer uses a simple serial protocol to communicate to the PC. This is illustrated below.

EEPROM Programmer - Serial Protocol

Command Byte	Parameters	Returns	Notes
A	Address Low Address High	Byte at supplied address	Reads from address supplied, acknowledges internally to allow further sequential reads
B	None	Byte at next address	Sequential read
C	Address Low Address High Data Byte	Acknowledgement (K character) after 10mS	Writes byte to supplied address
D	None	K	Confirms programmer present

There are 4 commands sent from the PC to the programmer, the first (A) reads from a defined address and leaves the programmer in a state to read the next byte sequentially, the second command (B) reads the next byte sequentially, the third command (C) writes a byte to a supplied address. Finally therefore is a simple command (D) which forces the programmer to return a 'K' character, this confirms that the programmer is present.

C Program for the PIC

The Program for the PIC is designed for the 16F84, it is included in the EEPROM Prog subdirectory of the Projects directory.

The listing is shown below:

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

const int SERIAL_RATE=9600; // Set serial
port rate
const int BITTIME_IN=PROCFREQ*1000/SERIAL_RATE/4;
const int BITTIME_OUT=PROCFREQ*1000/SERIAL_RATE/4;

const int SERIALPORT_IN=&PORTA; // Port
for serial i/f
const int SERIALBIT_IN=2; // Bit for
serial i/f

const int SERIALPORT_OUT=&PORTA; // Port
for serial i/f
const int SERIALBIT_OUT=3; // Bit for
seiral i/f

const BYTE PICIO=&PORTA;
const BYTE _SDA=1;
```

```

const BYTE _SCL=0;

BYTE AddrLo;    // EEPROM address - Low word
BYTE AddrHi;    // EEPROM address - High word
BYTE EEData;    // Data read or written from device

void RxAddr();
void WriteAddress();

void main()
{
    PORTA=0xff;                // Set IIC bus bits high
    TRISA=~((1<<_SCL)|(1<<SERIALBIT_OUT));    // Drive Clock &
Serial Data
    IIRead(IISTOP);            // Terminate pending read

    pSerialOut('K');           // Transmit OK Character
    while(1)
    {
        switch(pSerialIn())
        {
            case 'A' : RxAddr();
                        WriteAddress();
                        PORTA|=(1<<_SCL);        // Set clock high for
start bit
                        IIWrite(0xA1,IISTART|IIACK);    // Control byte for a
read
            case 'B' : TRISA|=(1<<_SDA);
                        pSerialOut(IIRead(IIACK)); break;
            case 'C' : RxAddr();
                        EEData=pSerialIn();
                        WriteAddress();
                        IIWrite(EEData,IIACK|IISTOP);
                        Wait(10);
            case 'D' : pSerialOut('K'); break;
        }
    }
}

//
// Read an address to operate at in Lo-Hi form
//
void RxAddr()
{
    AddrLo=pSerialIn();        // Read low byte of address
    AddrHi=pSerialIn();        // Read high byte of address
}

//
// We generate a stop state first (to terminate a possible read
operation)
// Then we address the LC65 with the control byte 0xA0
// Finally we send both upper and lower bytes of the address
//

void WriteAddress()
{
    QuickStop();                // generate stop state on bus
    IIWrite(0xA0,IISTART|IIACK); // Inform LC65 we are about to send
an address
    IIWrite(AddrHi,IIACK);      // Write upper byte of address
    IIWrite(AddrLo,IIACK);      // Write lower byte of address
}

```

The intialisation routines at the top of the main function set the PIC pins on Port A to drive and receive for the serial port and IIC device. The initialisation for the IIC port simply does a read with a stop bit using the IIRead function. This is not strictly necessary, but is included in

case the PIC is reset without the EEPROM device also being reset (such as during a brown out), in this case the EEPROM may be in the middle of a transaction and requires the stop bit to terminate. The first action of the board is to send a K character to confirm operation.

The main loop is a simple switch statement which reads a command character and acts upon it.

The RxAddr() function reads an address in low high format.

The WriteAddress() function terminates any current operation and then writes an address to the EEPROM. It is used before reading from a specified address (command A), and before programming a byte to the EEPROM (command C).

PC Application Program

To use the programmer from the PC we shall examine a very simple QBASIC program which allows the EEPROM to be read or a file to be programmed to it. QBASIC is supplied as part of DOS.

Owing to lack of space for listings the program is minimal and can be much improved, it is presented here with just the essential features for reading and writing EEPROM's. The BASIC program is shown below, press key 1 to read the first 256 bytes of the EEPROM, press 2 to read the next 256 bytes (sequentially), and press 3 to write a file to the EEPROM. Note that the file to be written is presented as a series of bytes in ASCII decimal, one byte per line, the first byte is written to address 0, the second byte to address 1 etc. There is little commenting in the file, or within this article, the program is quite straightforward and QBASIC has comprehensive on-line help. This program is also supplied in the EEPROM Prog directory along with a short example hex file called Test.Hex.

Important Note. Please note that within this BASIC program the communications port is set to send 2 stop bits, this is important to allow the PIC program to undertake processing between received bytes.

```

DECLARE SUB WriteFile ()
DECLARE FUNCTION NeatHex$ (x!, places!)
DECLARE SUB waitkey ()
DECLARE SUB Read256 ()
DECLARE SUB ReadFirst ()
DECLARE SUB ReadNext (Num!)
OPEN "COM1:9600,N,8,2,RS,DS,BIN" FOR RANDOM AS #1

DIM SHARED Address

WHILE 1
  CLS
  PRINT "EEPROM Programmer": PRINT
  PRINT "1-Read 1st 256 bytes of EEPROM"
  PRINT "2-Read sequentially - next 256 bytes of EEPROM"
  PRINT "3-Write file to EEPROM"

  a$ = "": WHILE a$ = "": a$ = INKEY$: WEND
  IF (a$ = "1") THEN ReadFirst
  IF (a$ = "2") THEN Read256
  IF (a$ = "3") THEN WriteFile
  IF (ASC(a$) = 27) THEN STOP
WEND

```

```

FUNCTION NeatHex$ (x, places)
  a$ = HEX$(x)
  WHILE LEN(a$) < places: a$ = "0" + a$: WEND
  PRINT a$; " ";
END FUNCTION

SUB Read256
  CLS
  FOR i = 1 TO 16
    PRINT (NeatHex$(Address, 4)); ": "; : ReadNext (16): PRINT
  NEXT
  waitkey
END SUB

SUB ReadFirst
  Address = 1
  PRINT #1, "A"; CHR$(0); CHR$(0);
  WHILE LOC(1) < 1: WEND
  CLS
  PRINT "0000 : "; NeatHex$(ASC(INPUT$(LOC(1), #1)), 2);
  ReadNext (15)
  PRINT
  FOR i = 1 TO 15
    PRINT NeatHex$(Address, 4); ": "; : ReadNext (16): PRINT
  NEXT
  waitkey
END SUB

SUB ReadNext (Num)
  FOR i = 1 TO Num
    PRINT #1, "B";
    WHILE LOC(1) < 1: WEND
    PRINT NeatHex$(ASC(INPUT$(LOC(1), #1)), 2);
    Address = Address + 1
  NEXT
END SUB

SUB waitkey
  PRINT : PRINT "Press a key to continue"
  WHILE INKEY$ = "": WEND
END SUB

SUB WriteFile
  Address = 0
  CLS
  PRINT "Enter filename >"; : INPUT f$
  OPEN f$ FOR INPUT AS #2
  DO WHILE NOT EOF(2)
    LINE INPUT #2, a$
    IF a$ <> "" THEN
      IF (ASC(a$) >= 48) AND (ASC(a$) <= 57) THEN
        x = VAL(a$)
        PRINT #1, "C"; CHR$(Address MOD 256); CHR$(INT(Address /
256)); CHR$(x);
        WHILE LOC(1) < 1: WEND
        IF INPUT$(LOC(1), #1) <> "K" THEN PRINT "Error on Receive"
        PRINT "Programmed "; NeatHex$(x, 2); " to "; NeatHex$(Address,
4)
        Address = Address + 1
      END IF
    END IF
  LOOP
  CLOSE #2
  waitkey
END SUB

```

Further notes on using I2C EEPROMs

Please note that the circuit application has a brown out reset circuit, this was originally inserted because the board was intended for use as the basis of the ETI PIC BASIC series. In this application the EEPROM is nearly always being read, and glitchy I/O lines during power down caused corruption of the EEPROM. It is recommended that such a circuit should always be used.

Even with a brown out circuit it is possible for corruption and failure to occur if the power supply disappears during a write operation. In important applications it is recommended that EEPROM data should be protected with a checksum, or by writing data 3 times - a vote being taken to decide on the correct data.

There are a number of other EEPROM devices with I²C interfaces which may be used, they are similar, and can be driven with little change to the application program.

7 *Optimising your output*

Introduction to Optimisation

Optimising Variables

Optimising Loops

Optimisation options

Optimising Variables and functions

Optimising Pointers

Introduction to optimisation

This chapter describes in brief, how to write C Code which results in the most efficient assembly code possible.

Optimising Variables

The PIC is an 8 bit microcontroller, and therefore the most efficient type of variable is the char type. Unsigned char types are more efficient than signed for some operations. All of the header files supplied with FED PIC C define the type BYTE which is an unsigned char type.

Optimisation 1

Use the BYTE type for all variables which may hold values between 0 and +255.

Use the char type for all variables which may hold values between -128 and +127.

Optimising Loops

The PIC has a special loop instruction DECFSZ. The compiler detects a specific type of loop and translates it to this special instruction type, such loops are very compact and fast. The form of the loop is as shown:

```
for(var=expr; var; var--)
```

var is signed or unsigned char type. For example the following code toggles a bit on PORTB 8 times:

```
BYTE i;
BYTE Mask=8;

TRISB=0;
for(i=8; i; i--)
{
    PORTB^=Mask;
    PORTB^=Mask;
}
```

Although it takes some practice to write loops in this fashion the saving in code and time is worth the extra effort.

Optimisation 2

Use the following loop form for efficient loops:

for(i=expression; i; i--)

where i is a variable which is of type signed, or unsigned char

Optimisation options

For the 14 bit core controllers use the PIC call stack provided that no more than 6 levels of function call are to be used (5 if interrupts are in use, many more for the enhanced range depending on the specific devices). For the 16 bit core controllers (18Fxxx series) use the PIC call stack provided that no more than 20 levels of function call are in use – up to 125 with the most recent devices, (in practice this allows nearly all 18Fxxx core programs to use the PIC call stack).

Use the Optimise for Space option unless extra speed is essential.

Set the number of Local Optimise Bytes to a reasonably large number, 16 is recommended on larger processors, 4 or 8 on smaller processors.

Optimisation 3

You can set the options for compilation optimisation by using the tab on the compiler options box which is shown when the project is compiled - when F9 is pressed or the Compile | Compile menu option is used. Use the Optimisations Tab, select Optimise for Space and Use PIC Call Stack (Quick Call). Finally set the number of Local Optimise Bytes to 4 (or greater if you can afford it).

Optimising Variables and functions

Some processors have multiple pages of RAM or ROM (such as the 16C558 which has two pages of RAM, or the 16C74 which has two pages of RAM and two pages of ROM). On these processors access to the lower page is faster than the higher pages.

Optimisation 4

With the larger PIC processors place the more often used global variables and functions earliest in the program, or use the register keyword to force into the lower page.

Optimising Pointers

FED PIC C allows the use of 1 or 2 byte pointers. 1 byte pointers may be used whenever an item is in the bottom 256 bytes of PIC address space. To define a pointer as one byte then include the keyword "ram" in the definition :

```
BYTE x;
BYTE ram *xp;
xp=&x;           // set variable x to 9
*xp=9;
```

1 byte (ram) pointers are very much more efficient than normal pointers, and can be used in the same way as any other pointer provided that the item is within the lower 256 bytes of RAM address.

For the 18Cxxx and 18Fxxx series of devices the ram pointer is still used to force the compiler to use FSR as a pointer accumulator, this considerably improves efficiency for accesses using those variables.

Optimisation 5

When the PIC has only two RAM pages (i.e. the top byte of RAM is at address hex 0xff or lower, then use RAM pointers at all times to point to items in RAM. If it has more than one page then use RAM pointers when you know that the item to which a pointer is directed is in the bottom 256 bytes of RAM.

Hint

If you want to use RAM pointers for the 16F877 or other devices with more than 2 pages of RAM then set the Stack Pointer to the value 0FF (use the Memory tab when the project is compiled). Now the stack and all variables will be in the bottom pages of RAM.

Hint

If you want to make your program portable to other compilers whilst still using RAM pointers then use the following code at the top of your program:

```
#ifndef _FEDPICC
#define ram
#endif
```


8 *Using Assembler*

Introduction to assembler

Simple use of Assembler

The FED PIC C Programmers model

Example of use of assembler

Example of use of assembler (2)

Macro Reference

Compiler sub-routine Reference

Assembler Projects (Professional Version only)

Introduction to assembler

It is quite straightforward to use Assembler within an FED PIC C program. At its simplest level it is possible to use assembler instructions directly within functions, more complex use such as writing complete C functions with assembler language requires a deeper understanding of the operation of FED PIC C.

Simple use of Assembler

To insert assembler mnemonics within a program simply use the **#pragma asm** and **#pragma asmend** directives (note that **#endasm** may also be used to be compatible with other compilers). All lines following a **#pragma asm** directive will be inserted directly into the assembler code (after normal macro expansion and other C pre-processor directives have been obeyed).

Example

The code below inserts a CLRWDI instruction into the function. This instruction clears the watchdog timer.

```
#pragma asm
CLRWDI          ; Clear the watchdog timer
#pragma asmend
```

An alternative if there is only one line of assembler code to be inserted is to use the **#pragma asmline** directive. In this case only one line of information is passed to the assembler - the code on the line following the directive. Note that if there is a comment on the line it should be preceded by a semi-colon (;) as this will also be passed to the assembler.

Another method for use in C source code, is the one used by Microchip, this resembles a macro :

```
asm("movlw 7");
```

Example

The code below inserts a CLRWDI instruction into the function, all three examples do the same. This instruction clears the watchdog timer.

```
#pragma asmline CLRWDT ; Clear the watchdog timer
#asmline CLRWDT ; Does the same
asm("clrwdt"); ; Compatible with other compilers
```

The FED PIC C programmers model

Memory Map

Memory Organisation

Common Functions and program operation

Use of ... form for function parameters

Memory Map

FED PIC C has a number of variables which are used in the operation of the program. These are known as the **Program Variables**. These variables (not shown in order) are as follows:

Name	Length	Notes
ACC	2 or 4	This is the internal accumulator. It is 2 bytes long (to allow for pointers), or is 4 bytes long if the program makes any use of long or float types. The Accumulator is used for holding intermediate results, and for returning values from functions
ACC2	2 or 4	This is the secondary accumulator which is used by internal routines. For example to multiply two numbers together ACC is loaded with the first number, ACC2 with the second, and then the multiply routine called, the result is stored in ACC
Temp	1	A temporary variable used by internal routines
Temp2	1	A temporary variable used by internal routines
Flags	1	A single byte used by internal routines
Heap	0 or 1	A pointer to the heap, only used if the heap is used. The heap is used to initialise complex objects (structures and arrays), and to pass complex objects back from functions.
HeapU	0 or 1	Upper byte of heap pointer used with processors with more than 256 bytes of RAM
PCLATHS	0 or 1	Used on PIC's with 2 or more pages of ROM to save the wanted value of PCLATH prior to a jump
Sp	1	The software stack pointer used to implement the software stack. This pointer always points to the first free byte on the stack. It is only used for simple 14 bit core models, the medium 14 bit model uses FSR 1 for stack, the 18 series uses FSR 2 for stack.
IntSave	0-3	Registers used to save interrupt status if interrupts are in use (on some processor models) (W, FSR and STATUS)
SaveInt	0+	Area used for saving program variables during an interrupt. This area is zero length when interrupts are not used, and is also zero length when Quick Interrupts are in use. If normal interrupts are in use then it is large enough to save most of the system variables.
IntFlags	0+	1 byte used on some PIC's to store information through a call.
LocOpt	0+	Memory bytes used for local variables in place of stack to optimise program operation. See Optimising your output .

At a minimum 8 bytes of memory are used in addition to any memory used by the program. The location of these variables depends on the processor type and is shown in the [Memory Organisation](#) section. When **long** variables are used together with the heap and interrupts then 16 bytes of memory are used in addition to that used by the program and the locally optimised variables [Optimising your output](#).

It is possible to refer to any of these variables by name within assembler routines. The following program includes an assembler function which simply returns the value 7 – it loads 7 into the accumulator and returns, the other lines in this assembler function are described in the following sections:

```
int Return7();
int x=-1;

void main()
{
    x=Return7();
endit:
    while(1);
}

#pragma asm
module "Return7"
Return7::
    movlw 7
    movwf ACC
    clrf ACC+1
    MRET 0
endmodule
#pragma asmend
```

Memory Organisation

The program variables, stack, heap and variables are stored in different locations on different processors. This section shows how the various elements are stored on different processors.

14 bit core

PAGE 0		PAGE 1	
0	System Variables	8	System Variables
12	Program Variables	0	
	Memory Variables		
	Heap ↓		

R	Stack↑
---	--------

All addresses are shown in hex. R is the top address in memory, the stack starts at the top address in memory and moves downwards as items are pushed on to it.

14 bit core and Enhanced Mid-range Core

PAGE 0		PAGE 1	
0	System Variables	80	System Variables
20	Program Variables	A0	Memory Variables
	Memory Variables		Heap ↓
		R	Stack↑

All addresses are shown in hex. R is the top address in memory, the stack starts at the top address in memory and moves downwards as items are pushed on to it.

Note that the stack cannot drop below address A0 as this requires a jump in memory (which would take more program space and time), so the stack is limited to the upper page. Similarly the heap always starts at address A0 even if every byte is not used by memory variables in the lower page. It is possible to set SP into the lower memory page using the Compile Options dialog box on the Memory tab, or by using the `#stack` directive. For the Enhanced Mid Range the Stack pointer uses FSR1 which is set in linear memory.

14 bit core

PAGE 0		PAGE 1		PAGE 2		PAGE 3	
0	System Variables	80	System Variables	0	System Variables	180	System Variables
20	Program Variables	A0	Memory Variables	200	Memory Variables		Heap ↓
	Memory Variables					R	Stack↑

70	Program Variables	F0	Program Variables	170	Program Variables	1F0	Program Variables
----	-------------------	----	-------------------	-----	-------------------	-----	-------------------

All addresses are shown in hex. R is the top address in memory, the stack starts at the top address in memory and moves downwards as items are pushed on to it. Note that some of the program variables are stored at address 0x70 which is repetitive throughout RAM and appears in each memory page.

Note that the stack cannot drop below address 180 as this requires a jump in memory (which would take more program space and time), so the stack is limited to the upper page.

16 bit core

PAGE 0 (0 to FF)		PAGE 1 - N-1		Last Page		Systems Var	
0	System Variables	100	Memory Variables	0500	Heap ↓	F80	PIC control registers
20	Program Variables			20	Stack ↑		
	Memory Variables						

All addresses are shown in hex. Note that the last page is the last RAM page, for the 18C452 this is shown from 500H to 5FFH. If a stack of more than 256 bytes is required then the heap can be manually relocated to a lower page using the [Compiler Options Dialog Box](#).

Common Functions and program operation

Introduction to common functions

Using subroutines in assembler code

Calling C functions from assembler

Use of Enhanced Mid Range processors

Notes for use of PCLATH and RP0,RP1 in the STATUS register

Notes for use of paging in 16 bit core devices

Notify the compiler of an assembler function

The Stack and Function calling

Complete C functions in assembler

Defines in FED PIC C

Introduction to common functions

The FED PIC C compiler generates code as efficiently as possible, however some operations require the use of subroutines which are called. All of the compiler subroutines are stored in the bottom page of ROM, and only those needed are included.

These subroutines may be used by assembler routines, but the compiler must be informed about their use as it has to include them in the final hex file.

The common functions are described at the end of this chapter.

Using subroutines in assembler code

All internal routines are in the bottom memory page. For 14 bit core devices (16xxx etc.), the linker directive SETPCLATH sets the PCLATH register back to the bottom page and must be used before any macros if the function could be located in higher ROM pages.

To tell the compiler to include the correct routines use the **#pragma asmdefine** (or **#pragma asmdefine**) pre-processor directive. Open the file "Bit14_nK.asm" which is in the libs sub-directory of the C Compiler. This includes all of the routines for the C Compiler. Search for the label CopyMem, in this file - it should look like this:

```

ifdef _CopyMem
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Copy W bytes from address in PACC to address in PACC2, preserve
PACC2
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

CopyMem
    _LARAM=1
    _PCLW=1
    _Push2D=1
    _PopD=1
    _Pop2D=1

    movwf Temp2          ; Save count
    call Push2D          ; Save PACC2

```

Note that the CopyMem function will only be included if the definition _CopyMem is included in the file. To do this use the #pragma asmdefine function as shown here in this example:

```

#pragma asmdefine _CopyMem

#pragma asm
    movlw x
    movwf ACC
    clrf ACC+1
    movlw y
    movwf ACC2
    clrf ACC2+1
    movlw 4
    call CopyMem
#pragma asmend

```

This example copies 4 bytes from the address x to the address y. If x and y were long variables it would copy x to y. Here is a complete example program which will work with 16 or 18 Series devices:

```
void cp();
long x,y;

void main()
{
    x=0x12345678;
    cp();
    while(1);
}

void cp()
{
    #pragma asmdefine _CopyMem

    #pragma asm
    movlw x
    movwf ACC
    clrf ACC+1
    movlw y
    movwf ACC2
    clrf ACC2+1
    movlw 4
    call CopyMem
    #pragma asmend
}
```

In practice this is not as onerous a task as it may appear at first. Most assembler routines will use only one or two compiler subroutines - otherwise the C code may as well be used directly.

Calling C functions from assembler

If an assembler or C library function is to be called within a different module, then the #callfunction macro must be used. E.g. If the functions Delays in a different C file is to be called then use :

```
#callfunction Delays
```

To call another C function the macro MCALL must be used.

This will probably only affect advanced users.

Use of Enhanced Mid Range processors ()

The enhanced mid range processors are not compatible with the standard 14 bit core processors. FED have rewritten all the library assembler routines to support the Enhanced Mid Range. However in so doing we have tried to maintain the same macro names as for the standard range so that existing code will work as much as possible. For example MSETIRP will set the IRP to the value of the supplied argument in the standard core, for the Enhanced Mid Range it will set the value of FSROH to the supplied argument. Similarly any macros which set RP0 and RP1 will set BSR instead for the Enhanced Mid Range.

For the Enhanced Mid Range the stack uses linear addressing space and is set to the top of memory, the heap is also set into linear space. There is no sp variable, instead FSR1 is used as the stack pointer and always points to the first free address on the stack.

Notes for use of PCLATH and RP0,RP1 (& BSR) in the STATUS register

This section **only applies to 14 bit core processors** with two or more ROM pages or two or more RAM pages.

The IRP bit, FSR the accumulators (ACC & ACC2), Temp, Temp2, and Flags may be changed at will within a function. The return value from a function is stored in ACC or ACC2.

ROM Paging - PCLATH

When the function is called...

When a function is called PCLATH is set to the address of the function. PCLATH must be cleared before any macros are used within the function, as the C core functions are all located in the bottom memory page.

The SETPCLATH or SETPCLATHA special Linker instructions may be used to set PCLATH if the processor has more than 1 page of ROM.

The syntax of the SETPCLATH instruction is as follows:

SETPCLATH Expression[,CurrentValue][,CanClr]

The first parameter – Expression is the value of the memory address for which PCLATH has to be modified. The other two parameters are optional. The CurrentValue parameter is supplied if the current value of PCLATH is known, if it is unknown it may be left out, or set to –1. The CanClr parameter is set if the linker is permitted to use the clr instruction to clear PCLATH which will set the Z flag. The linker will set PCLATH if it needs to (for example if Expression and CurrentValue are in the same page then PCLATH may be left as it stands). Therefore this instruction may result in zero, one or two words of hex code. Here is an example from the Data library – the start of the SerialIn function:

```
SerialIn::
    SETPCLATH 0,SerialIn,1
    if _QUICKCALL==1
        MGETFSRSPO 1          ; FSR points to Count
    else
        MGETFSRSPO 3          ; FSR points to Count
    endif
    movfw 0
    movwf ACC2                ; PORT to ACC2
```

Note that the SETPCLATH instruction needs to set the memory page to 0 where all the common functions reside, and that the current value is known to be SerialIn as this is the label which has just been called.

The SETPCLATHA instruction is similar, but has the following syntax:

SETPCLATHA Expression

This instruction always takes exactly two words in the hex file (even for processors with only one page) and is essential for software delay loops where the exact code length must be guaranteed.

Whether a C function is written in assembler (see [Complete C functions in assembler](#)), or as inline assembly within a C function it, then it will use modules. This will guarantee that all the code in the function is within a single memory page.

When a function returns...

The following simple rules are followed - when QuickCall is set then PCLATH is always cleared before returning from a function. When QuickCall is clear then PCLATH is always set to the address of the module which called the function when the function returns. It will always be cleared from a call to a common function in the base area.

This may sound complicated but in practice is very simply achieved by using the MRET macro. This has the following syntax:

```
MRET 0
```

The parameter is ignored for the 14 bit core series and should always be supplied as 0.

Example

Consider the following simple example program which includes a function called func, which takes a parameter (N) and increments PORTB N times. It makes use of the instructions discussed in this section. Note that this function is written in assembler as a linker module and therefore the whole function will always be in a single ROM page. Please also note the use of #pragma asmfunc which tells the compiler that func is an assembly function and that parameters must not be optimised on to the stack.

```
#include <pic.h>
void func(unsigned char x);
#pragma asmfunc func

void main()
{
    PORTB=0;
    TRISB=0;
    func(7);
endit:
    while(1);
}

#pragma asm

    module "func"
func::

    SETPCLATH 0,func,1 ; Clear PCLATH (if set)
    if _QUICKCALL==1   ; Test quickcall option
        MGETFSRSPO 1   ; FSR points to Count
    else
        MGETFSRSPO 3   ; FSR points to Count
    endif

    movfw 0             ; Pick up the count
    movwf ACC           ; Use ACC as a counter

    SETPCLATH func,0,1 ; Set PCLATH back to func
```

```

funcLoop:
    incf PORTB
    decfsz ACC
    goto funcLoop

    MRET 0                ; Return

#pragma asmend

```

RAM Paging

In normal operation the RAM paging bits (RP0 and RP1, or BSR for the Enhanced Mid Range) are set to 0. They may be adjusted in a function but must be reset to 0 before returning from a function. There are two macros of use for the RAM paging bits MSETRP and MCLEARRP. They both take a memory address. MSETRP sets RP0 and RP1 (or BSR) to the correct address, MCLEARRP clears the bits if they were set by the MSETRP macro.

For example the following code clears the TRISB register and sets the paging bits back to 0 afterwards.

```

MSETRP TRISB
clrf TRISB
MCLEARRP TRISB

```

This will work for the standard and Enhanced Mid Range cores.

FSR operations need to take account of the IRP bit. IRP (or the FSROH register for the Enhanced Mid Range) may be set to any value when a function is called, and does not have to be cleared on return..

Notes for use of paging in 16 bit core devices

This section only applies to 16 bit core processors.

The BSR may be changed at will within user defined assembler functions, but should be saved through in line assembler within C functions.

ROM Paging

The PCLATH/PCLATU registers may be changed at will. It is strongly suggested that user functions should always use the linker instructions SMARTCALL or SMARTJUMP instead of CALL or GOTO when a label is not known. These instruction have a single parameter that is the address to be called, the linker will insert an rcall or call, or bra or goto dependent on the distance of the jump.

The linker manual shows the other SMART instructions which may be used with the 16 series.

Examples:

```

SMARTCALL GetSPOW

SMARTJUMP 4

```

Complete C functions in assembler

It is possible to write complete C functions in assembler – several examples have already been given. Assembler functions must use the linker, and must be defined as modules. Please see the linker manual for full details (which is online in the menu option **Help | Assembler Help**, the linker manual is a top level link from this file).

The module is defined using the MODULE directive. For an assembler function the only relevant option is the name of the function which is defined in inverted commas :

```
module "MyFunction"
```

Now all labels in the function will be local except for those which are followed by a double colon (::), these will be global. For any C function this will only ever be the first label, which is the function address:

```
MyFunction::
```

After this the function may be written as described above, and then the module must be terminated with an endmodule directive :

```
endmodule
```

For most assembler functions the programmer will not want the duplicate block optimiser to “attack” the code, especially during debugging. For this reason the module line should be followed by the line :

```
dupmodoff
```

and prior to the end module the duplicate optimiser can be turned back on again :

```
dupmodon
```

If the source file only contains assembler functions then only one dupmodoff needs to be given at the top of the first module.

As described above the best method of returning is to use the MRET macro which will automatically set PCLATH (for 14 bit core processors) and return using the internal or software stack dependant on the compiler options.

Here is the complete function MyFunction() clears all the ports of the processor.

```
#pragma asm
module "MyFunction"

dupmodoff          ; Turn off optimiser, ignored by assembler

MyFunction::
    movlw PORTA
    movwf FSR        ; PORTA address to FSR
    if RAMPAGES>2
        bcf STATUS,IRP    ; Clear IRP if necessary
    endif

    movlw 5
    movwf ACC        ; Use ACC as a counter

ClearLoop:
    clrf 0
```

```

    incf FSR
    decfsz ACC
    goto ClearLoop

    MRET 0                ; return

dupmodon                ; Turn on optimiser

endmodule
# pragma asmend

```

Defines in FED PIC C

Any constant integer in FED PIC C will be converted to a value in the assembler routines with an underscore prefixed. Thus

```
const int Port=6;
```

is converted to the following in the assembler file:

```
_Port=6
```

Thus your assembler routine may use `_Port` in the routine and it will be replaced by the value 6. This also applies to the constants which are automatically defined in FED PIC C (See [Defines](#)).

Use of ... form for function parameters

The ... form for function parameters is supported in FED PIC C – but there are no macros provided to decompose the parameters, this must be undertaken in assembler. A function using ... must be declared as an assembler function, this is achieved with the `asmfunction` preprocessor directive:

```
#pragma asmfunction printf
```

This ensures that parameters will always be passed on the stack. Now the item at the top of the stack (the last item pushed before the return address) will be a single byte value which is the number of parameters pushed. This is followed by the fixed parameters, then the optional parameters.

Optional parameters are pushed as integer (2 byte) or long (4 byte) values. Characters and 8 bit values are pushed as integers and always appear on the stack as 2 bytes.

For an example written in assembler see the `fnprintf` function in the `strings.c` library file.

Notify the compiler of an assembler function

The compiler must be notified if a function is written entirely in assembler. This is to stop any attempt to optimise its local variables or parameters. To do this use the following directive:

```
#pragma asmfunc FunctionName
```

For example the following definition is used within the header file for the library assembler function `WriteEEData`:

```
void WriteEEData(unsigned char Addr,unsigned char Data);
#pragma asmfunc WriteEEData
```

Now the parameters for the assembler function will always be pushed on the stack. The following section shows how to read them.

The Stack and Function calling

The stack is implemented in software as the PIC stack cannot be used to save and recall data. The stack pointer is called sp, it points to the first free byte on the stack. For the Enhanced Mid Range there is no sp, the stack pointer is held in FSR1. Within 16 bit core devices there is no sp variable - the stack pointer is held within FSR2. Throughout this section sp should be replaced with FSR 1 for Enhanced Mid Range, or FSR2 when considering 16 bit core devices.

Consider the stack on the 16C74. The figure below shows the empty stack, sp holds the value 0xFF:

	Address	Contents
Stack Pointer ➔	0xFF	-
	0xFE	-
	0xFD	-

Now if the integer value (2 bytes) 0x1234 is pushed on to the stack it will change and sp now holds the value 0xFD as follows:

	Address	Contents
	0xFF	12
	0xFE	34
Stack Pointer ➔	0xFD	-

Finally if the single byte 0xFD is popped from the stack then sp will hold the value 0xFE, and the stack is:

	Address	Contents
	0xFF	12
Stack Pointer ➔	0xFE	34
	0xFD	-

Now if a function is called, the parameters for the function are stored on the stack. If the compiler is not set to use Quick Calling (see [Optimising Your Output](#)), then the return address from the function is also stored on the stack. If the compiler is using Quick Calling (which is the default), then the return address is stored on the PIC's internal call stack.

Parameters for a function are pushed on to the stack from right to left. Consider the following C code:

```
unsigned char func(int x,char y);
.
.
.
func(1000,-1);
```

The dots show that there is other C code in here. Now when the function is called from within the initial function (main) the stack will look like this:

	Address	Contents	Notes
	0xFF	0xFF	Parameter y
	0xFE	0x03	Upper byte parameter x
	0xFD	0xE8	Lower byte parameter x
	0xFC	RH	Return address upper byte
	0xFB	RL	Return address lower byte
Stack Pointer ➔	0xFA	-	

If the compiler is using Quick Calling then the stack will look like this

	Address	Contents	Notes
	0xFF	0xFF	Parameter y
	0xFE	0x03	Upper byte parameter x
	0xFD	0xE8	Lower byte parameter x
Stack Pointer ➔	0xFC	-	Return address on PIC stack

So it is possible to read a parameter from the stack using assembler by using the stack pointer sp. For example using Quick Calling the following code will read the value of parameter y into the W register:

```

movlw 3          ; Offset of parameter y from sp
addwf sp,w       ; Move address of parameter y to W
movwf FSR        ; Move to FSR
movfw 0          ; and read value of y indirectly

```

If we are not using Quick Calling then the first line would replace "**movlw 3**" by "**movlw 5**" as there are another 2 bytes on the stack.

There is another method of achieving the same effect using a macro (note all macros are defined in the Macro Reference).

```

#pragma asmdefine _Getspo      // Needed for use of MGETFSRSPO

MGETFSRSPO 3                  ; Get address of y to FSR
movfw 0                        ; and read value of y indirectly

```

Example of use of assembler

We'll look in detail at the complete development of a C function written in assembler for the 16 series (after this example we'll show it again for the 18 series).

Consider a function to address an 8 bit port as a bi-directional data port. It will need to read or write data and will have two control signals – a read strobe and a write clock. The read and write signals will stay low for at least 1uS whatever the clock rate of the processor. The function we will use to drive the port will have the following template which could be included in a header file.

```

unsigned char Drive8(char Read,unsigned char *vp);
#pragma asmfunc Drive8

```

The function will either write the value addressed by the pointer vp, or read the value from the port into that variable. The flag Read will be set if bus is to be read, and reset if it is to be written. The returned value will be the contents of the variable addressed by vp whether it has been read or written. Note the use of asmfunc to stop the compiler from optimising parameters in calls to the function.

Establishing a 1uS delay

Firstly we need to look at establishing the delay macro which will insert at least 1uS delay with faster clock rates. Look at the following code:

```

ClockDelay      equ    D'1000'
FreqNS          equ    (D'1000000000'/_APROCFREQ)*4      ; Cycle period in
nS
LowCyc          equ    ClockDelay/FreqNS

```

Clock Delay is the time we need to wait in nano-Seconds, FreqNS will be the cycle period in nano-seconds (the time in nS required to execute one instruction). Finally LowCyc will be the number of cycles that we need to keep the read or write strobes low.

Now we can write a macro to insert the number of cycles (with nop instructions) required to ensure at least 5uS delay:

```

STROBETIME macro exdelay

```

```

dx=exdelay
while dx>0
    NOP
    dx-=1
endw
endm

```

Note that for slow clock rates the LowCyc time will be 0, and therefore the macro will insert no NOP instructions.

Setting up the port and control bits

We need to allow the user of this function to define the 8 bit port for bi-directional data, and also the port and bits for the read and write signals. We can do this by using constant integers and we can choose any names we like. We'll use the following – and the example shows the C code required to set up the function to use PORTD as the 8 bit port and Port C bits 0 and 1 as the read and write signals respectively.

```

const int DataPort8=&PORTD;
const int ReadStrobePort=&PORTC;
const int ReadStrobeBit=0;
const int WriteStrobePort=&PORTC;
const int WriteStrobeBit=1;

```

Recall that in assembler we will now have five constants defined as follows :

```

_DataPort8
_ReadStrobePort
_ReadStrobeBit
_WriteStrobePort
_WriteStrobeBit

```

Writing the function

We'll look at the code first, and then take a look at the wrapper to turn it into a module.

The first thing we need to do is to get the parameters and store them, we'll store the flag to define read or write into the system variable ACC, and the address pointer vp into FSR:

```

Drive8:: ; Function label (global)

        bsf _WriteStrobePort,_WriteStrobeBit
        bsf _ReadStrobePort,_ReadStrobeBit
        bsf STATUS,RP0
        bcf _WriteStrobePort,_WriteStrobeBit
        bcf _ReadStrobePort,_ReadStrobeBit
        bcf STATUS,RP0

        SETPCLATH 0,Drive8 ; Set PCLATH to common functions
if _QUICKCALL==1
    MGETFSRSPO 1           ; Set FSR to point to 1st parameter
else
    MGETFSRSPO 3           ; (When software stack is used)
endif
        movfw 0
        movwf ACC          ; First parameter (ReadFlag) to ACC
        incf FSR
        movfw 0
        movwf ACC+1        ; Temporary store
        incf FSR
        movfw 0            ; Upper byte of the address
pointer

```



```

bcf STATUS,IRP
skpz
bsf STATUS,IRP
movfw ACC+1
movwf FSR
SETPCLATH Drive8,0 ; Set PCLATH back to this module

```

This code is straightforward. Bear in mind that functions have RP0 and RP1 set to the lowest page before they are called so that we can address the system variables (including ACC) without changing them. The first action we undertake is to set the read and write strobe bits high, and the TRIS bits for those lines low so that they drive a high value.

The MGETFSRSPO macro gets the stack pointer offset to FSR so that FSR points to the parameters for the function. A test is undertaken so that if _QUICKCALL is set (the PIC calling stack is used) then the parameters will be offset by 1 byte instead of 3. The ReadFlag is read and stored in the ACC variable. The variable pointer (vp) lower byte is read, stored in ACC+1 temporarily, and then the upper byte is read and used to set IRP. Finally the lower byte is stored back in FSR so that FSR now addresses our variable.

The next action is to test whether we are to read or write. The following code fragment tests the flag and the section which writes to the bus is shown:

```

movf ACC,f ; Test the read flag
skpz
goto Read8 ; Jump forward if Read

movfw 0 ; Read the variable
movwf _DataPort8 ; Write to the port
bsf STATUS,RP0 ; Point to Upper bank
clrf _DataPort8 ; Drive data port
bcf STATUS,RP0 ; Back to lower page

bcf _WriteStrobePort,_WriteStrobeBit

STROBETIME LowCyc ; Delay strobe

bsf _WriteStrobePort,_WriteStrobeBit

bsf STATUS,RP0 ; Point to Upper bank
decf _DataPort8 ; Read data port again
bcf STATUS,RP0 ; Back to lower page

goto Drive8End ; Go to the end of the
function

```

We simply write the variable to the port and then set the port to drive. The write strobe is asserted low for 1uS and then the port set to drive again.

Finally we'll take a look at the wrapper around the function – the read section will not be presented in detail, but shown as part of the overall program.

Here is the wrapper :

```

#pragma asm
#pragma asmdefine _GetSPotoFSR0

module "Drive8"

; Code here

Drive8End: ; End of function

```

```

        movfw 0
        movwf ACC          ; Return value

        MRET 0             ; Macro to return

endmodule

#pragma asmend

```

The line with the `asmdefine` is used to tell the compiler to include a function in the common area – see the table in the [Macro Reference](#) section below. Note that the return value from the function is stored in ACC. The macro `MRET` executes a return using the PIC stack or the software stack dependant on the use of quick calling.

Here is the complete function together with a test program which can be assembled and run:

```

#include <pic.h>

unsigned char Drive8(char Read,unsigned char *vp);
#pragma asmfunc Drive8

const int DataPort8=&PORTD;
const int ReadStrobePort=&PORTC;
const int ReadStrobeBit=0;
const int WriteStrobePort=&PORTC;
const int WriteStrobeBit=1;

BYTE x,y,z;

void main()
{
    x=0x55;
    z=Drive8(0,&x); // Write 0x55 to Port B
    z=Drive8(1,&y); // Read the port to y and to z

endit:
    while(1);
}

#pragma asm

    ClockDelay    equ    D'1000'
    FreqNS equ     (D'1000000000'/_APROCFREQ)*4      ; Cycle period in
nS
    LowCyc equ     ClockDelay/FreqNS

STROBETIME macro exdelay
    dx=exdelay
    while dx>0
        NOP
        dx-=1
    endw
endm

module "Drive8"

dupmodoff ; Turn off optimiser, ignored by assembler

Drive8:: ; Function label (global)

    bsf _WriteStrobePort,_WriteStrobeBit
    bsf _ReadStrobePort,_ReadStrobeBit
    bsf STATUS,RP0
    bcf _WriteStrobePort,_WriteStrobeBit

```

```

        bcf _ReadStrobePort,_ReadStrobeBit
        bcf STATUS,RP0

        SETPCLATH 0,Drive8 ; Set PCLATH to common functions
    if _QUICKCALL==1
        MGETFSRSP0 1 ; Set FSR to point to 1st parameter
    else
        MGETFSRSP0 3 ; (When software stack is used)
    endif
    movfw 0
    movwf ACC ; First parameter (ReadFlag) to ACC
    incf FSR
    movfw 0
    movwf ACC+1 ; Temporary store
    incf FSR
    movfw 0 ; Upper byte of the address
pointer
    bcf STATUS,IRP
    skpz
    bsf STATUS,IRP
    movfw ACC+1
    movwf FSR
    SETPCLATH Drive8,0 ; Set PCLATH back to this module

    movf ACC,f ; Test the read flag
    skpz
    goto Read8 ; Jump forward if Read

    movfw 0 ; Read the variable
    movwf _DataPort8 ; Write to the port
    bsf STATUS,RP0 ; Point to Upper bank
    clrf _DataPort8 ; Drive data port
    bcf STATUS,RP0 ; Back to lower page

    bcf _WriteStrobePort,_WriteStrobeBit

    STROBETIME LowCyc ; Delay strobe

    bsf _WriteStrobePort,_WriteStrobeBit

    bsf STATUS,RP0 ; Point to Upper bank
    decf _DataPort8 ; Read data port again
    bcf STATUS,RP0 ; Back to lower page

    goto Drive8End ; Go to the end of the
function
Read8: bcf _ReadStrobePort,_ReadStrobeBit

    STROBETIME LowCyc ; At least 1uS

    movfw _DataPort8
    movwf 0 ; Save to variable

    bsf _ReadStrobePort,_ReadStrobeBit

Drive8End: ; End of function
    movfw 0
    movwf ACC ; Return value

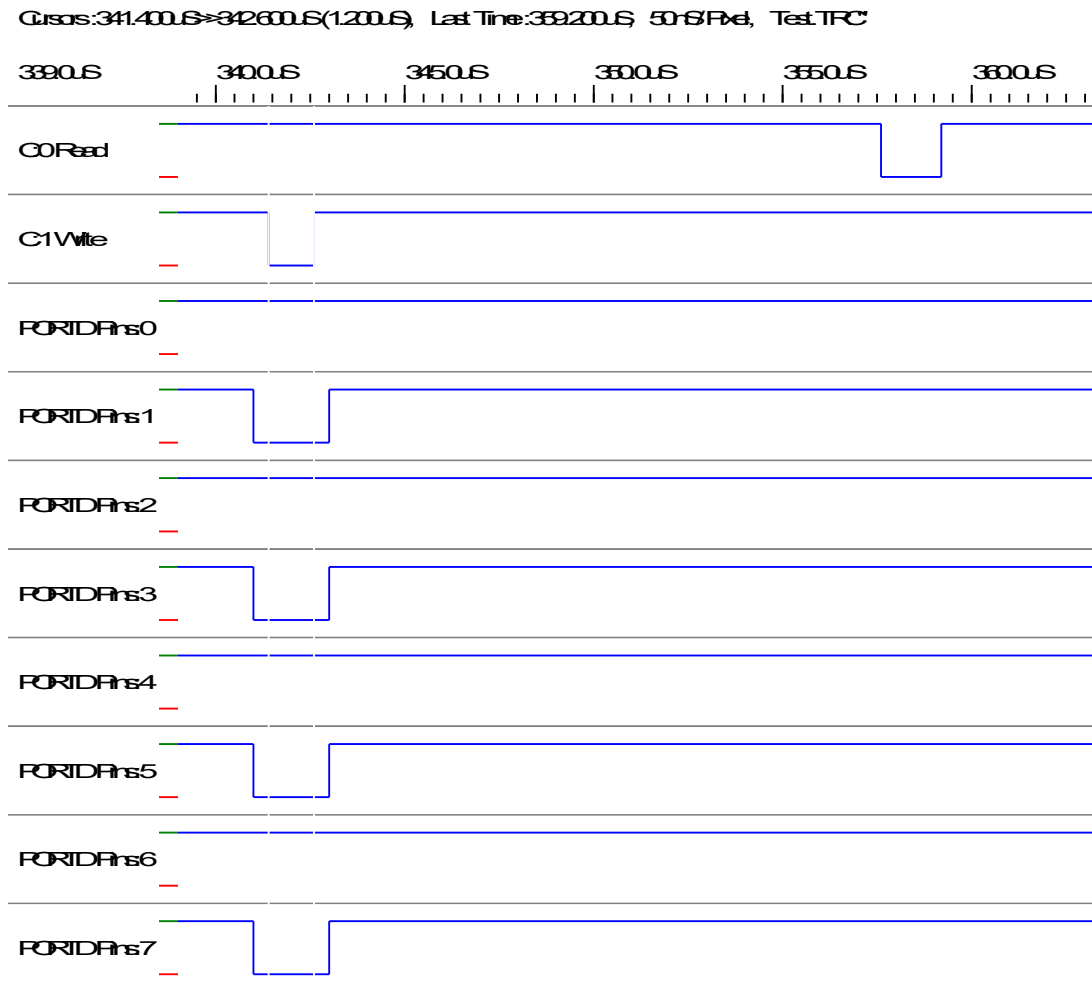
    MRET 0 ; Macro to return

dupmodon ; Turn optimiser back on

#pragma asmend

```

Here is the waveforms shown by this simple test program using a 20MHz clock – as can be seen the write and read strobes are of the correct width:



Here is the version for the 18 series :

```
#pragma asm

#pragma asmdefine _GetSP0toFSR0

ClockDelay equ D'1000'
FreqNS equ (D'1000000000'/_APROCFREQ)*4 ; Cycle period in
nS
LowCyc equ ClockDelay/FreqNS

STROBETIME macro exdelay
    dx=exdelay
    while dx>0
        NOP
        dx-=1
    endw
endm

module "Drive8"

dupmodoff ; Turn off optimiser, ignored by assembler
```

```

Drive8:: ; Function label (global)

        bsf _WriteStrobePort,_WriteStrobeBit
        bsf _ReadStrobePort,_ReadStrobeBit
        bcf _WriteStrobePort+0x12,_WriteStrobeBit
        bcf _ReadStrobePort+0x12,_ReadStrobeBit

        if _QUICKCALL==1
            MGETFSRSP0 1 ; Set FSR to point to 1st parameter
        else
            MGETFSRSP0 3 ; (When software stack is used)
        endif
        movf POSTINC0,w
        movwf ACC ; First parameter (ReadFlag) to ACC
        movf POSTINC0,w
        movwf FSR1L ; Address in FSR1
        movf INDF0,w ; Upper byte of the address pointer
        movwf FSR1H

        tstfsz ACC ; Test the read flag
        bra Read8 ; Jump forward if Read

        movf INDF1,w ; Read the variable
        movwf _DataPort8 ; Write to the port
        clrf _DataPort8+0x12 ; Drive data port

        bcf _WriteStrobePort,_WriteStrobeBit

        STROBETIME LowCyc ; Delay strobe

        bsf _WriteStrobePort,_WriteStrobeBit

        setf _DataPort8+0x12 ; Read data port again
        bra Drive8End ; Go to the end of the function

Read8: ; bcf _ReadStrobePort,_ReadStrobeBit

        STROBETIME LowCyc ; At least 1uS

        movf _DataPort8,w
        movwf INDF1 ; Save to variable

        bsf _ReadStrobePort,_ReadStrobeBit

Drive8End: ; End of function
        movff INDF1,ACC ; Return value

        MRET 0 ; Macro to return

dupmodon ; Turn optimiser back on

endmodule

#pragma asmend

```

Example of use of assembler (2)

Here is the commented assembler function for the library function :

```
void Wait(unsigned int WaitmS);
```

This function waits for the specified number of milliseconds.

```
#ifndef _Wait
#define _LoadSPD
#pragma asm

dupmodoff ; Turn off optimiser for rest of file

        module "_Wait"
#define _Delays          ; Needed to use _DTIME macro

//////////
//////////
;
; void Wait(unsigned int Delayms)
;
//////////
//////////

Wait::
    SETPCLATH 0,Wait,1
    if _QUICKCALL==1
        movlw 1
        call LoadSPD
    else
        movlw 3
        call LoadSPD
    endif
    incf FSR
    SETPCLATH _WL0
    call _WL0
    clrf PCLATH
    MRET 0

if ROMPAGES>1
_DTIME=_APROCFREQ/4000-10
else
_DTIME=_APROCFREQ/4000-8
endif

_WL0::
    movf ACC,w          ; 1
    iorwf ACC+1,w        ; 1
    btfsc STATUS,Z              ; 2
    return
    DELAY _DTIME          ; Macro to delay exactly _DTIME
cycles
    SETPCLATHA _WL0      ; 0/2 (Dependant on processor)
    decf ACC             ; 1
    incfsz ACC,w         ; 1
    goto _WL0            ; 2
    decf ACC+1           ; 1
    goto _WL0            ; 2

        endmodule

#pragma asmend
#endif
```

Macro Reference

Please note that some of the macros require a knowledge of the operation of the compiler and are not useful within user defined assembler routines. They are provided here as a reference for experienced programmers or to trace operation of the core library routines, and FED cannot provide more detail on their operation than provided here.

Macro	Parameters	Size	Flags	Notes
MCALL	N	6		Call C function with label or address n To use this macro you must use the following definition: #pragma asmdefine _PushRA
MCLEARRP	N	0-2		Clear RP0 and/or RP1 if they would have been set to address location N. Or sets BSR for the Enhanced Mid Range.
MGETFSRSPO	n	2		Load PACC with address of Stack offset n To use this macro you must use the following definition: #pragma asmdefine _Getspo For the 18 series you must use the following : #pragma asmdefine _GetSPotoFSR0
MRET	N	1+		Return from C function - note the parameter should always be 0 (1 is used for interrupts)
MSETIRP	Addr	0-1		Set IRP to point to the correct page for Addr. No effect on processors with 2 or less pages of RAM. For the Enhanced Mid Range sets FSR0H.
MSETIRPSP		0-1		Set IRP to point to the same page as sp. No effect on processors with 2 or less pages of RAM. Sets FSR0H to FSR1H for the Enhanced Mid Range.
MSETRP	Address	0-2		Set RP0 and RP1 to address if required (initially RP0 and RP1 are assumed to be 0). For the Enhanced Mid Range sets BSR.

Compiler sub-routine Reference

Any of the subroutines in the compiler general purpose library can be called from assembler. Most require the use of `#asmdefine`. They are all shown together with the `#asmdefine` statement required to use them. Note 16 and 32 bit values are held in memory least significant byte first.

Subroutine	How to define	Notes
AddA	<code>#asmdefine _AddA</code>	Add single byte in ACC to ACC2 - result in ACC
AddD	<code>#asmdefine _AddD</code>	Add 16 bit integer in ACC to ACC2 - result in ACC
AddQ	<code>#asmdefine _AddQ</code>	Add 32 bit long integer ACC to ACC2 - result in ACC
AndA	<code>#asmdefine _AndA</code>	Bitwise And single byte in ACC to ACC2 - result in ACC
AndD	<code>#asmdefine _AndD</code>	Bitwise And 16 bit integer in ACC to ACC2 - result in ACC
AndQ	<code>#asmdefine _AndQ</code>	Bitwise And 32 bit long integer ACC to ACC2 - result in ACC
BitNotD	<code>#asmdefine _BitNotD</code>	Bitwise invert all bits of 16 bit integer in ACC
BitNotQ	<code>#asmdefine _BitNotQ</code>	Bitwise invert all bits of 32 bit integer in ACC
CheckQ	<code>#asmdefine _CheckQ</code>	Set zero flag if 32 bit integer in ACC is zero
ClearHeap	<code>#asmdefine _Clearheap</code>	Clear the heap
CompSA	<code>#asmdefine _CompSA</code>	Compare 8 bit signed integer in ACC with ACC2 - set flags (Z and C)
CompSD	<code>#asmdefine _CompSD</code>	Compare 16 bit signed integer in ACC with ACC2 - set flags (Z and C)
CompSD	<code>#asmdefine _CompSD</code>	Compare 16 bit signed integer in ACC with ACC2 - set flags (Z and C)
CompSQ	<code>#asmdefine _CompSQ</code>	Compare 32 bit signed integer in ACC with ACC2 - set flags (Z and C)
CompUSD	<code>#asmdefine _CompUSD</code>	Compare 16 bit unsigned integer in ACC with ACC2 - set flags (Z and C)
CompUSQ	<code>#asmdefine _CompUSQ</code>	Compare 32 bit unsigned integer in ACC with ACC2 - set flags (Z and C)
ConvGT	<code>#asmdefine _ConvGT</code>	Convert flags Z and C which result from a PIC subtraction operation (subwf) into a Z if a GT operation is true
ConvGTE	<code>#asmdefine _ConvGTE</code>	Convert flags Z and C which result from a PIC subtraction operation

Subroutine	How to define	Notes
		(subwf) into a Z if a >= operation is true
ConvLT	#asmdefine _ConvLT	Convert flags Z and C which result from a PIC subtraction operation (subwf) into a Z if a < operation is true
ConvLTE	#asmdefine _ConvLTE	Convert flags Z and C which result from a PIC subtraction operation (subwf) into a Z if a <= operation is true
ConvNEQ	#asmdefine _ConvNEQ	Convert flags Z and C which result from a PIC subtraction operation (subwf) into a Z if a != operation is true
Copy2A	#asmdefine _Copy2A	Copy Acc2 to Acc (8 bits)
Copy2D	#asmdefine _Copy2D	Copy Acc2 to Acc (16 bits)
Copy2Q	#asmdefine _Copy2Q	Copy Acc2 to Acc (32 bits)
CopyA	#asmdefine _CopyA	Copy Acc to Acc2 (8 bits)
CopyD	#asmdefine _CopyD	Copy Acc to Acc2 (16 bits)
CopyMem	#asmdefine _CopyMem	Copy W bytes from address in ACC to the address in ACC2. Note address to copy from can be in ROM in which case bit 15 is set.
CopyQ	#asmdefine _CopyQ	Copy Acc to Acc2 (32 bits)
CopyROMHeap	#asmdefine _CopyROMHeap	Copy W bytes from ROM to heap address in PACC2
DecD	#asmdefine _DecD	Decement 16 bits in ACC
DecQ	#asmdefine _DecQ	Decement 32 bits in ACC
DivA	#asmdefine _DivA	Signed Divide 8 bits in ACC by ACC2 - result in ACC
DivD	#asmdefine _DivD	Signed Divide 16 bits in ACC by ACC2 - result in ACC
DivQ	#asmdefine _DivQ	Signed Divide 16 bits in ACC by ACC2 - result in ACC
DivUSA	#asmdefine _DivUSA	Unsigned Divide 8 bits in ACC by ACC2 - result in ACC
DivUSD	#asmdefine _DivUSD	Unsigned Divide 16 bits in ACC by ACC2 - result in ACC
DivUSQ	#asmdefine _DivUSQ	Unsigned Divide 16 bits in ACC by ACC2 - result in ACC
DoSwitch	#asmdefine _DoSwitch	Do a switch table, Temp and PCLATH will hold address, ACC holds the switch value (8 to 32 bits), ACC2 holds nitems (16 bit), Temp2 holds length of comparison
Getspo	#asmdefine _Getspo	Get stack pointer offset in W to FSR
GetSPO	#asmdefine _GetSPO	Get stack pointer offset in W to an address in ACC
GetSPO2	#asmdefine _GetSPO2	Get stack pointer offset in W to an address in ACC2
GetspoW	#asmdefine _GetspoW	Get stack pointer offset by W bytes to FSR

Subroutine	How to define	Notes
GotoPACC	#asmdefine _GotoPACC	Goto 16 bit address in ACC
IncD	#asmdefine _IncD	Increment 16 bits in ACC
IncQ	#asmdefine _IncQ	Increment 32 bits in ACC
LeftShiftA	#asmdefine _LeftShiftA	Shift ACC (8 bits) left by number of bits specified in ACC2
LeftShiftD	#asmdefine _LeftShiftD	Shift ACC (16 bits) left by number of bits specified in ACC2
LeftShiftQ	#asmdefine _LeftShiftQ	Shift ACC (32 bits) left by number of bits specified in ACC2
LoadField	#asmdefine _LoadField	Load a bit field. Temp holds the number of bits. W holds the starting bit number. The Address is in ACC. 16 or 32 bit result in ACC.
LoadFSRA	#asmdefine _LoadWA	Move 8 bits from address in FSR to ACC
LoadFSRD	#asmdefine _LoadWD	Move 16 bits from address in FSR to ACC
LoadFSRQ	#asmdefine _LoadWQ	Move 32 bits from address in FSR to ACC
LoadHeap2A	#asmdefine _LoadHeap2A	Load 8 bits from heap offset in W to ACC2
LoadHeap2D	#asmdefine _LoadHeap2D	Load 16 bits from heap offset in W to ACC2
LoadHeap2Q	#asmdefine _LoadHeap2Q	Load 32 bits from heap offset in W to ACC2
LoadHeapA	#asmdefine _LoadHeapA	Load 8 bits from heap offset in W to ACC
LoadHeapD	#asmdefine _LoadHeapD	Load 16 bits from heap offset in W to ACC
LoadHeapLabel	#asmdefine _LoadHeapLabel	Load address of heap offset in W to ACC (16 bits)
LoadHeapLabel2	#asmdefine _LoadHeapLabel2	Load address of heap offset in W to ACC2 (16 bits)
LoadHeapQ	#asmdefine _LoadHeapQ	Load 32 bits from heap offset in W to ACC
LoadPACCA	#asmdefine _LoadPACCA	Load ACC (8 bits) from address in ACC (16 bits). Note if top bit is set (bit 15) then an address in ROM will be used
LoadPACCD	#asmdefine _LoadPACCD	Load ACC (16 bits) from address in ACC (16 bits). Note if top bit is set (bit 15) then an address in ROM will be used
LoadPACCQ	#asmdefine _LoadPACCQ	Load ACC (32 bits) from address in ACC (16 bits). Note if top bit is set (bit 15) then an address in ROM will be used
LoadROM2A	#asmdefine _LoadROM2A	Load ACC2 (8 bits) from ROM address in W and in PCLATHS (16 bits).
LoadROM2D	#asmdefine _LoadROM2D	Load ACC2 (16 bits) from ROM address in W and in PCLATHS (16 bits).

Subroutine	How to define	Notes
LoadROM2Q	#asmdefine _LoadROM2Q	Load ACC2 (32 bits) from ROM address in W and in PCLATHS (16 bits).
LoadROMA	#asmdefine _LoadROMA	Load ACC (8 bits) from ROM address in W and in PCLATHS (16 bits).
LoadROMD	#asmdefine _LoadROMD	Load ACC (16 bits) from ROM address in W and in PCLATHS (16 bits).
LoadROMQ	#asmdefine _LoadROMQ	Load ACC (32 bits) from ROM address in W and in PCLATHS (16 bits).
LoadSP2A	#asmdefine _LoadSP2A	Load ACC2 (8 bits) from Stack offset in W
LoadSP2D	#asmdefine _LoadSP2D	Load ACC2 (16 bits) from Stack offset in W
LoadSP2Q	#asmdefine _LoadSP2Q	Load ACC2 (32 bits) from Stack offset in W
LoadSPA	#asmdefine _LoadSPA	Load ACC (8 bits) from Stack offset in W
LoadSPD	#asmdefine _LoadSPD	Load ACC (16 bits) from Stack offset in W
LoadSPQ	#asmdefine _LoadSPQ	Load ACC (32 bits) from Stack offset in W
LoadWD	#asmdefine _LoadWD	Load ACC (16 bits) with contents of memory addressed by W, and IRP bit
LoadWD2	#asmdefine _LoadWD2	Load ACC2 (16 bits) with contents of memory addressed by W, and IRP bit
LoadWQ	#asmdefine _LoadWQ	Load ACC (32 bits) with contents of memory addressed by W, and IRP bit
LoadWQ2	#asmdefine _LoadWQ2	Load ACC2 (32 bits) with contents of memory addressed by W, and IRP bit
LSEQ	#asmdefine _ConvEQ	Convert flags Z and C which result from a PIC subtraction operation (subwf) into a 0 or 1 in ACC for a == operation
LSGT	#asmdefine _ConvGT	Convert flags Z and C which result from a PIC subtraction operation (subwf) into a 0 or 1 in ACC for a > operation
LSGTE	#asmdefine _ConvGTE	Convert flags Z and C which result from a PIC subtraction operation (subwf) into a 0 or 1 in ACC for a >= operation
LSLT	#asmdefine _ConvLT	Convert flags Z and C which result from a PIC subtraction operation (subwf) into a 0 or 1 in ACC for a < operation
LSLTE	#asmdefine _ConvLTE	Convert flags Z and C which result from a PIC subtraction operation (subwf) into a 0 or 1 in ACC for a <= operation

Subroutine	How to define	Notes
		<= operation
LSNEQ	#asmdefine _ConvNEQ	Convert flags Z and C which result from a PIC subtraction operation (subwf) into a 0 or 1 in ACC for a != operation
ModA	#asmdefine _ModA	Signed Modulus (remainder of division) 8 bits in ACC by ACC2 - result in ACC
ModD	#asmdefine _ModD	Signed Modulus (remainder of division) 16 bits in ACC by ACC2 - result in ACC
ModQ	#asmdefine _ModQ #asmdefine _ExtACC	Signed Modulus (remainder of division) 16 bits in ACC by ACC2 - result in ACC
ModUSA	#asmdefine _ModUSA	Unsigned Modulus (remainder of division) 8 bits in ACC by ACC2 - result in ACC
ModUSD	#asmdefine _ModUSD	Unsigned Modulus (remainder of division) 16 bits in ACC by ACC2 - result in ACC
ModUSQ	#asmdefine _ModUSQ #asmdefine _ExtACC	Unsigned Modulus (remainder of division) 16 bits in ACC by ACC2 - result in ACC
MulA	#asmdefine _MulA	Signed Multiply 8 bits in ACC by ACC2 - result in ACC
MulD	#asmdefine _MulD	Signed Multiply 16 bits in ACC by ACC2 - result in ACC
MulQ	#asmdefine _MulQ #asmdefine _ExtACC	Signed Multiply 16 bits in ACC by ACC2 - result in ACC
MulUSA	#asmdefine _MulUSA	Unsigned Multiply 8 bits in ACC by ACC2 - result in ACC
MulUSD	#asmdefine _MulUSD	Unsigned Multiply 16 bits in ACC by ACC2 - result in ACC
MulUSQ	#asmdefine _MulUSQ #asmdefine _ExtACC	Unsigned Multiply 16 bits in ACC by ACC2 - result in ACC
NegA	#asmdefine _NegA	Negate 8 bit value in ACC
NegD	#asmdefine _NegD	Negate 16 bit value in ACC
NegQ	#asmdefine _NegQ	Negate 32 bit value in ACC
OrA	#asmdefine _OrA	Bitwise Or single byte in ACC to ACC2 - result in ACC
OrD	#asmdefine _OrD	Bitwise Or 16 bit integer in ACC to ACC2 - result in ACC
OrQ	#asmdefine _OrQ	Bitwise Or 32 bit long integer ACC to ACC2 - result in ACC
Pop2A	#asmdefine _Pop2A	Pop 8 bit value to ACC2 from the stack
Pop2D	#asmdefine _Pop2D	Pop 16 bit value to ACC2 from the stack
Pop2Q	#asmdefine _Pop2Q	Pop 32 bit value to ACC2 from the stack
PopA	#asmdefine _PopA	Pop 8 bit value to ACC from the stack
PopD	#asmdefine _PopD	Pop 16 bit value to ACC from the

Subroutine	How to define	Notes
		stack
PopQ	#asmdefine _PopQ	Pop 32 bit value to ACC from the stack
Push2A	#asmdefine _Push2A	Push 8 bit value in ACC2 to the stack
Push2D	#asmdefine _Push2D	Push 16 bit value in ACC2 to the stack
Push2Q	#asmdefine _Push2Q	Push 32 bit value in ACC2 to the stack
PushA	#asmdefine _PushA	Push 8 bit value in ACC to the stack
PushD	#asmdefine _PushD	Push 16 bit value in ACC to the stack
Pushn	#asmdefine _PushA #asmdefine _GotoPACC	Push 8 bit value in W to the stack and also copy to ACC
PushQ	#asmdefine _PushQ	Push 32 bit value in ACC to the stack
PushW	#asmdefine _PushW	Push W onto stack
RightShiftA	#asmdefine _RightShiftA	Shift ACC (8 bits) right by number of bits specified in ACC2 (signed)
RightShiftD	#asmdefine _RightShiftD	Shift ACC (16 bits) right by number of bits specified in ACC2 (signed)
RightShiftQ	#asmdefine _RightShiftQ	Shift ACC (32 bits) right by number of bits specified in ACC2 (signed)
RightShiftUSA	#asmdefine _RightShiftUSA	Shift ACC (8 bits) right by number of bits specified in ACC2 (unsigned)
RightShiftUSD	#asmdefine _RightShiftUSD	Shift ACC (16 bits) right by number of bits specified in ACC2 (unsigned)
RightShiftUSQ	#asmdefine _RightShiftUSQ	Shift ACC (32 bits) right by number of bits specified in ACC2 (unsigned)
SaveField	#asmdefine _SaveField	Save a bit field. Temp holds the number of bits. W holds the starting bit number. The Address is in ACC. Value in ACC.
SaveFSRA	#asmdefine _PushA	Save 8 bits from ACC to address in FSR
SaveFSRD	#asmdefine _SaveFSRD	Save 16 bits from ACC to address in FSR. Note FSR points to the upper byte first and ends up pointing to the lower byte.
SaveFSRnD	#asmdefine _SaveFSRnD	Save 16 bits from ACC to address in FSR. Note FSR points to the lower byte first and ends up pointing to the upper byte.
SaveFSRnQ	#asmdefine _SaveFSRnQ	Save 32 bits from ACC to address in FSR. Note FSR points to the lower byte first and ends up pointing to the upper byte.
SaveFSRQ	#asmdefine _SaveFSRQ	Save 32 bits from ACC to address

Subroutine	How to define	Notes
		in FSR. Note FSR points to the upper byte first and ends up pointing to the lower byte.
SavePACC2A	<code>#asmdefine _SavePACC2A</code>	Save 8 bits from ACC to the memory address specified in ACC2 (16 bits)
SavePACC2D	<code>#asmdefine _SavePACC2D</code>	Save 16 bits from ACC to the memory address specified in ACC2 (16 bits)
SavePACC2Q	<code>#asmdefine _SavePACC2Q</code>	Save 32 bits from ACC to the memory address specified in ACC2 (16 bits)
Sex2AD	<code>#asmdefine _Sex2AD</code>	Sign extend ACC2 (8 bits) to ACC2 (16 bits)
Sex2AQ	<code>#asmdefine _Sex2AQ</code>	Sign extend ACC2 (8 bits) to ACC2 (32 bits)
Sex2DQ	<code>#asmdefine _Sex2DQ</code>	Sign extend ACC2 (16 bits) to ACC2 (32 bits)
SexAD	<code>#asmdefine _SexAD</code>	Sign extend ACC (8 bits) to ACC (16 bits)
SexAQ	<code>#asmdefine _SexAQ</code>	Sign extend ACC (8 bits) to ACC (32 bits)
SexDQ	<code>#asmdefine _SexDQ</code>	Sign extend ACC (16 bits) to ACC (32 bits)
SubA	<code>#asmdefine _SubA</code>	Subtract single byte in ACC from ACC2 - result in ACC
SubD	<code>#asmdefine _SubD</code>	Subtract 16 bit integer in ACC from ACC2 - result in ACC
SubQ	<code>#asmdefine _SubQ</code>	Subtract 32 bit long integer ACC to ACC2 - result in ACC
SwapA	<code>#asmdefine _SwapA</code>	Swap ACC and ACC2 - 8 bits
SwapD	<code>#asmdefine _SwapD</code>	Swap ACC and ACC2 - 16 bits
SwapQ	<code>#asmdefine _SwapQ</code>	Swap ACC and ACC2 - 32 bits
SwapTosA	<code>#asmdefine _SwapTosA</code>	Swap ACC (8 bit) with the 8 bit value on the Top of the Stack
SwapTosD	<code>#asmdefine _SwapTosD</code>	Swap ACC (16 bit) with the 16 bit value on the Top of the Stack
SwapTosQ	<code>#asmdefine _SwapTosQ</code>	Swap ACC (32 bit) with the 32 bit value on the Top of the Stack
UsexAQ	<code>#asmdefine _UsexAQ</code>	Unsigned extend 8 bit ACC to 32 bit ACC
UsexDQ	<code>#asmdefine _UsexDQ</code>	Unsigned extend 16 bit ACC to 32 bit ACC
XOrA	<code>#asmdefine _XOrA</code>	Bitwise XOr single byte in ACC to ACC2 - result in ACC
XOrD	<code>#asmdefine _XOrD</code>	Bitwise XOr 16 bit integer in ACC to ACC2 - result in ACC
XOrQ	<code>#asmdefine _XOrQ</code>	Bitwise XOr 32 bit long integer ACC to ACC2 - result in ACC
ZLTE	<code>#asmdefine _ZLTE</code>	Set Z flag if result of last PIC subtraction would result in a <= comparison being true

9 *Interrupts & Memory*

Interrupts

Normal Interrupts

Quick Interrupts

High Priority Interrupts

Allocation of memory

Interrupts

There are two types of interrupt, Normal Interrupts and Quick Interrupts. Normal interrupts allow most C functions to be used, quick interrupts are much more restrictive, but use much less program and file register memory. **It is recommended that Quick Interrupts are always used where possible.**

It is also recommended that as little as possible is undertaken within the Interrupt routine - detect the interrupt source, set a flag, and test and operate on the flag in the main program. However if it is essential to undertake operations within an Interrupt routine for speed then it is recommended that they are written in assembler.

Note that using quick interrupts then C functions called from within an interrupt routine are always called using the PIC call instruction - therefore **within an interrupt routine you cannot call normal C functions**, but only assembler functions.

Normal Interrupts

The FED PIC C compiler supports interrupts which are available on all the processors supported by the compiler. The interrupt function is a void function with no parameters and should be called Interrupt. The interrupt function will be called automatically every time that a PIC interrupt occurs. It is important that the Interrupt function tests and clears the flags which caused the interrupt, however the GIE flag is automatically set by the compiler.

Normal interrupts have quite a big overhead on PIC memory but they do allow the full use of C constructs, local variables, and any C function can be freely called from an interrupt function. Interrupt latency is quite high with a normal interrupt owing to the need to push the C status on the stack (or save it in the interrupt area).

To define a normal interrupt then the following constant integer should be defined within one of the C source files:

```
const int NormalInt=1;
```

Assembler and Normal Interrupts

Within the interrupt routine for a Normal Interrupt it is safe to alter any of the C system variables (ACC, ACC2 etc), FSR, BSR, W and STATUS (which are all saved). For the 18 series FSR0 and FSR1 may be freely altered without saving them (of course FSR2 is used for the software stack), also PCLATH, PRODL, PRODH, and the TBLPTR registers are also saved automatically by the compiler. For the 12/16 series the interrupt status is saved in an area of memory called SaveInterrupt, for the 18 series the status is saved on the stack.

Here is a very simple example program for the 16F84, it may be found in the Interrupt1 directory. Timer 0 is set to operate on the internal clock divided by 16 and so it will overflow every 4.096mS with a 4MHz clock. The variable x is incremented once every 4mS by the Timer0 interrupt caused when Timer 0 overflows.

```
//
// Simple interrupt demonstration, increment variable x and set
// variable Flag
// once every 4.096mS
//

#include <pic.h>

void Interrupt();

BYTE Flag;           // Flag set when interrupt has passed
BYTE x;              // Variable which is incremented

void main()
{
    TMR0=0;           // Clear timer 0
    OPTION_REG=(1<<NOT_RBPU) | (1<<PS1) | (1<<PS0); // Timer 0
    internal, div by 16
    INTCON=(1<<GIE) | (1<<T0IE); // Enable Timer 0
    interrupts

    while(1)
    {
        if (Flag) {Flag=0; x++;} // 4mS has passed, increment x
    }

    const int NormalInt=1;

    void Interrupt()
    {
        if (INTCON&(1<<T0IF)) // Test TMR0 interrupt flag
        {
            INTCON&=~(1<<T0IF); // Clear TMR0 interrupt flag
            Flag=1; // Set flag to show 4mS has passed
        }
    }
}
```

Quick Interrupts

The FED PIC C compiler supports interrupts which are available on all the processors supported by the compiler. The interrupt function is a void function with no parameters and should be called Interrupt. The interrupt function will be called automatically every time that a PIC interrupt occurs. It is important that the Interrupt function tests and clears the flags which caused the interrupt, however the GIE flag is automatically set by the compiler.

Quick Interrupts are only permitted to use very simple C instructions and may not use local variables. These are the following operations provided that the variables used on the left of the operation are unsigned char types, and on the right are constants:

```
if
&=
|=
&
=
```

Thus the following may all be used in a quick interrupt routine

```
if (PIR1&(1<<CMIE))
PIE1&=~(1<<CMIF);
Flag=1;
Flag|=2;
```

Provided that Flag is an unsigned char. Note that PIR1 and PIE1 are defined in the header file as file registers.

Assembler functions may be freely used in an interrupt routine, but please do not change the compiler memory areas (e.g. ACC). For the 16 series W and FSR are saved during an interrupt routine and may be freely changed. For the 18 series BSR, W and STATUS are saved, any use of FSR0 and FSR1 is prohibited unless they are also saved and restored by the assembler routine.

Thus the interrupt routine should simply check the interrupt flags and clear them, and set flag registers which can be tested in the main program. This is good practice in any event as undertaking significant processing within interrupt routines risks missing further interrupts.

Quick Interrupts is the default and no special action needs to be taken to use them. Previous versions of the compiler mandated the definition of the constant integer QuickInt, the necessity for this has been removed, but the definition may be left intact.

Here is the interrupt routine shown above again, but using quick interrupts, this is in the Interrupt2 directory.

```
//
// Simple interrupt demonstration, increment variable x and set
// variable Flag
// once every 4.096mS
//

#include <pic.h>

void Interrupt();

BYTE Flag;           // Flag set when interrupt has passed
BYTE x;              // Variable which is incremented

void main()
{
    TMR0=0;           // Clear timer 0
    OPTION_REG=(1<<NOT_RBPU)|(1<<PS1)|(1<<PS0); // Timer 0
    internal, div by 16
    INTCON=(1<<GIE)|(1<<T0IE); // Enable Timer 0
    interrupts

    while(1)
    {
        if (Flag) {Flag=0; x++;} // 4mS has passed, increment x
    }
}

void Interrupt()
{
    if (INTCON&(1<<T0IF)) // Test TMR0 interrupt flag
    {
        INTCON&=~(1<<T0IF); // Clear TMR0 interrupt flag
        Flag=1; // Set flag to show 4mS has passed
    }
}
```

```
}
```

As an indication of the efficiency of quick interrupts, it is interesting to note that the second version takes less bytes of RAM and less words of program memory.

High Priority Interrupts

High priority Interrupts are only permitted on the 18xxx series. The high priority interrupt routine is a void function called InterruptHi. The other restrictions on interrupts also apply to high priority interrupts – if normal interrupts are in use then full C constructs may be used in either the high or normal priority routine, for quick interrupts only the simpler source code may be used.

The compiler handles setting and clearing of GIEL, and GIEH.

Here is an example of a program with low priority and high priority interrupts:

```
#include <p18c452.h>

char IntCnt;
char IntCntHi;

void main()
{
    PIE1|=(1<<TMR1IE);          // Timer 1 interrupt
    INTCON|=(1<<GIEL)|(1<<GIEH); // Enable Hi & low priority
    RCON|=(1<<IPEN);             // 18Cxxx interrupt mode

    INTCON|=(1<<TMR0IE);          // Timer 0 interrupts
    INTCON2&=~(1<<TMR0IP);        // Timer 0 is low priority
    T0CON&=~(1<<T0CS);            // Timer 0 on internal
    T1CON|=(1<<TMR1ON);           // Timer 1 on
    T1CON&=~(1<<TMR1CS);         // Timer 1 on internal

endit:
    while(1);
}

void Interrupt()
{
    if (INTCON&(1<<T0IF)) INTCON&=~(1<<T0IF);
    IntCnt++;
}

void InterruptHi()
{
    if (PIR1&(1<<TMR1IF)) PIR1&=~(1<<TMR1IF);
    IntCntHi++;
}
```

Allocation of memory

The FED PIC C compiler does not include library routines for allocating memory, this is owing to the overhead of the structures required to keep track of memory allocations and freed areas which is not appropriate for PIC's which have so little RAM.

To allow manual allocation of memory it is possible to reserve space above the stack which will not be used by the compiler. This area can then be accessed by assigning variables and

locating them using the #locate compiler directive, or by direct access using addresses and pointers.

Here is an example program for the 16C558. The 16C558 has memory from 20H to 7FH and another memory page from A0H to BFH. Normally the stack pointer will be placed at location BFH. However it may be placed at 7FH. This can be done by using the Stack Pointer box in the Memory tab of the Compile Options Dialog box (which is brought up when the project is compiled). An alternative is to use the #STACK compiler directive which allows the stack to be assigned within the main file, this takes priority over the value set in the Compile Options Dialog box.

Here is an example program for the 16C558 which defines an array called arr of 16 integers from address A0H to BFH. The 1st element of the array is set to 1 and then incremented, in the first case by referencing array arr, and in the second by accessing through a direct memory address.

```
#include <P16C558.h>

#pragma stack 0x7F

extern int arr[16];
#pragma locate arr 0xA0

void main()
{
    arr[1]=1;
    arr[1]++;
    (*(int *)0xA2)++;

endit:
    while(1);
}
```

10 Creating Libraries

Introduction to Libraries

Including libraries in the FED PIC C environment

Library Examples

Introduction to Libraries

Libraries are automatically included by the header file. The pre processor allows library files to be included – for example here is the part of the datalib.h file which includes the correct libraries for the selected processor :

```
#if _CORE==16
#pragma wizcpp uselib "$(FEDPATH)\libs\commplib16.c"
#pragma wizcpp uselib "$(FEDPATH)\libs\datalib16.c"
#pragma wizcpp uselib "$(FEDPATH)\libs\delays16.c"
#else
#if _HASEXTINS
#pragma wizcpp uselib "$(FEDPATH)\libs\commlibx.c"
#pragma wizcpp uselib "$(FEDPATH)\libs\datalibx.c"
#pragma wizcpp uselib "$(FEDPATH)\libs\delaysx.c"
#else
#pragma wizcpp uselib "$(FEDPATH)\libs\commlib.c"
#pragma wizcpp uselib "$(FEDPATH)\libs\datalib.c"
#pragma wizcpp uselib "$(FEDPATH)\libs\delays.c"
#endif
#endif
#end
```

Note how three library files are included and there are three processor types, the `_CORE` test at the start if for the 16 bit, 18 series. The `_HASEXTINS` test is for the Enhanced Mid Range 14 bit processors.

Libraries on the FED PIC Compiler allow code to be written which will not be included if it is not used. In practice library functions are quite easy to write.

Every time that a function is called a special record of that function is created which can be tested using the `#ifdef` processor directive. The definition created may only be tested by using `#ifdef`, and has the name of the function preceded by an underscore. Thus if the program calls the function `SerialIn()`, then a definition `_SerialIn` is created. The C directive `#ifdef _SerialIn` will then return true, and subsequent code will be compiled.

This record lives through all files included in the C Compiler project. Therefore the library file(s) is/are included last in the project so that they are compiled after all other files. Then each library function is conditionally included by the use of `#ifdef` directives.

Including libraries in the FED PIC C environment

As standard a number of library files are included in FED PIC C, library files must be normal C files and should be placed in the Libs subdirectory of the PIC C Compiler. Similarly header files should also be included in this directory.

To add or remove a library file then use the File | Libraries menu option. This brings up the Set Libraries dialog box. n.

From version 12 onwards it is possible to include libraries automatically. There is a sub-directory of the Libs directory called “Auto”. Any file with a .C extension in this directory will be included as a library file automatically. This directory is also included in the search path for header files. **This is now obsolete and although it still works, should NOT be used – use the pre-processor (uselib) format at the start of this section**

Set Libraries Dialog Box

The set libraries dialog box allows new libraries to be added or removed. Click the Add button to bring up a file dialog to allow a new library file to be selected and added. Select a file in the library list and press Remove to delete a library file from the list. **This is now obsolete and although it still works, should NOT be used – use the pre-processor (uselib) format at the start of this section**

Library files are compiled in the order that they appear in the Current Libraries list.

Library Examples

Consider the following library C function called `vstrcpy(char *a, char *b)`, this is a void function which copies string b to string a:

```
#ifdef _vstrcpy

void vstrcpy(char *a, char *b)
{
    while(*b++=*a++);
}

#endif
```

11 Library Reference

BootLoader
ClockDataIn
ClockDataOut
Dallas 1 wire bus
EEPROM Routines
Graphic LCD functions
I2C Routines
IRDA IR Routines
RC5 IR Routines
Interrupt Driven Serial Port
isFunctions
Hex Keypad
LCD
LCDString
Maths Routines
mem functions
printf Functions
Random Numbers
SerialIn
SerialOut
stdio Functions
stdlib Functions
String Functions
String Print Functions
Wait

BootLoader

Specific functions defined in library;

Header file : None

Library file : <BootLoader.c>

These functions are provided in the BootLoader library. This library is not included in the compilation by default, but must be added using the File | Libraries menu option. For full details please see the separate manual BootLoader.pdf supplied in the Libraries sub-directory of the installation CD.

ClockDataIn

```
unsigned int ClockDataIn(char *Port,unsigned char Count,  
;           unsigned char ClockBitMask,unsigned char  
DataBitMask);
```

```
unsigned int pClockDataIn(unsigned char Count);
```

```
Header file : <datalib.h>
```

```
Library file : <datalib.c>
```

These routines are used to clock in serial data at an input PIN of the PIC. The entire words (specified from 1 to 16 bits) is then received and returned as an unsigned character. A clock pulse of a minimum of 1uS width is given by XORing the specified port with ClockBitMask twice, the input pin specified by DataBitMask is then read LSB first. Count bits are read (from 1 to 16) and the result returned as an unsigned integer (16 bits).

The first version of the routine allows for the user to define the port and the bit to be used to detect the received data. The second version sets up the port and the bit with constants which are defined in the C source file, and therefore this routine always operates on the same port, and the same bit. The second version is more useful when there is only one synchronous serial data port for input in use with the PIC, it also uses less space and is slightly faster.

The port and bits used for the pClockDataIn version are set up with the following constants:

```
CDI_PORT
```

```
CDI_CLOCKMASK
```

```
CDI_DATAMASK
```

See the examples to see how these are set up. Note that the clock pin must be set up as an output, and the data pin as an input.

Returns:

The received serial data.

Examples:

```
//  
// Example 1 - ClockDataIn()  
//  
  
//  
// Read a byte from Port B  
// bit 0 is clock, bit 1 is data  
//  
#include <pic.h>  
#include <datalib.h>  
  
unsigned char x;  
  
void main()  
{  
    PORTB=0;
```



```

    TRISB=0xfe;
    x=ClockDataIn(&PORTB,8,1,2);
    while(1);
}

//
// Example 2 - pClockDataIn()
//

//
// Read a byte from Port B
// bit 0 is clock, bit 1 is data
//
#include <pic.h>
#include <datalib.h>

const BYTE CDI_PORT=6;
const BYTE CDI_CLOCKMASK=1;
const BYTE CDI_DATAMASK=1;

unsigned char x;

void main()
{
    PORTB=0;
    TRISB=0xfe;
    x=pClockDataIn(8);
    while(1);
}

```

ClockDataOut

```

void ClockDataOut(char *Port,unsigned char Count,
                unsigned char ClockBitMask,unsigned char DataBitMask,
                unsigned int Data);

```

```

unsigned int pClockDataOut(unsigned char Count,unsigned int Data);

```

Header file : <datalib.h>

Library file : <datalib.c>

These routines are used to clock out serial data at an output PIN of the PIC. The entire work is then received and returned as an unsigned character. A clock pulse of a minimum of 1uS width is given by XORing the specified port with ClockBitMask, the output pin specified by DataBitMask is then set low or high according to the next data output bit - LSB first. Finally the clock mask is XOR'd with the port again. Count bits are transmitted (from 1 to 16).

The first version of the routine allows for the user to define the port and the bit to be used to detect the received data. The second version sets up the port and the bit with constants which are defined in the C source file, and therefore this routine always operates on the same port, and the same bit. The second version is more useful when there is only one synchronous serial data port for input in use with the PIC, it also uses less space and is slightly faster.

The port and bits used for the pClockDataIn version are set up with the following constants:

CDO_PORT

CDO_CLOCKMASK

CDO_DATAMASK

See the examples to see how these are set up. Note that the clock pin and data pins must be set up as outputs.

Returns:

Void function - no return.

Examples:

```
//
// Example 1 - ClockDataOut()
//

//
// Transmit a byte to Port B
// bit 0 is clock, bit 1 is data
//
#include <pic.h>
#include <datalib.h>

unsigned char x;

void main()
{
    x=0x55;
    PORTB=0;
    TRISB=0xfc;
    ClockDataOut(&PORTB,8,1,2,x);
    while(1);
}

//
// Example 2 - pClockDataOut()
//

//
// Read a byte from Port B
// bit 0 is clock, bit 1 is data
//
#include <pic.h>
#include <datalib.h>

const BYTE CDI_PORT=6;
const BYTE CDI_CLOCKMASK=1;
const BYTE CDI_DATAMASK=1;

unsigned char x;

void main()
{
    x=0x55;
    PORTB=0;
    TRISB=0xfc;
    ClockDataOut(8,x);
    while(1);
}
```

Dallas 1 Wire Bus

unsigned char ReadEEData(unsigned char Address);
char ResetCheck1W(void);
void Tx1Wire(unsigned char v);
unsigned char Rx1Wire(void);
Header file : <datalib.h>
Library file : <datalib.c>

The Dallas 1 wire interface provides an interface to the 1 wire bus used for "Memory Buttons" and other devices. Three routines are provided to reset the bus and check for devices on it, and to transmit and receive 8 bits at a time from the bus.

The full interface is described on the single wire interface web page at <http://www.ibutton.com/>

There is one I/O pin for the bus, this may be connected to any pin on the PIC. This pin is driven through the Tri-state register and therefore no external components bar the pull up resistor are required.

The I/O pin should be connected to Vdd with a 5K resistor in accordance with the 1 wire bus specification, note that if Port B pull ups are used then the internal resistor pull up is roughly equivalent to a 10K resistor so another 10K resistor should be connected externally.

To use the bus the reset routine (ResetCheck1W) should be called regularly (say about once every 20mS), this routine takes about 800uS to run. If a button is found then it can be interrogated (using the Tx1Wire and Rx1Wire routines), if a button is not found then the processor can run other tasks until checking again in another 20mS or so.

Interrupts should be disabled when using the bus routines.

To define the port used for the 1 bit bus then the constant integer TxRx1WirePort should be defined and set to the correct port, to define the bit on that port then the integer TxRx1WireBit should be defined and set to the port, the following shows how to set the port to PORTB, bit 1:

```
const int TxRx1WirePort=&PORTB;
const int TxRx1WireBit=1;
```

char ResetCheck1W(void);

Call this routine to reset the bus and check for devices on it. The routine returns with 0 if no device is present. It returns with 1 if a button is found, and with -1 if the bus is stuck low.

void Tx1Wire(unsigned char v);

This routine transmits the supplied byte (v) to the 1 bit bus.

unsigned char Rx1Wire(void);

This routine receives a byte from the bus and returns it.

EEPROM Routines

unsigned char ReadEEData(unsigned char Address);
void WriteEEData(unsigned char Address,unsigned char Data);
Header file : <datalib.h>
Library file : <datalib.c>

These routines are used to read and write the data EEPROM data within the 16C8x, 16F8x, and 16F87x devices.

To write a value to an EEPROM data location then use the WriteEEData function, the two parameters are the address, and the data value to write to that address. Note that the function does not wait for the write operation to complete before returning. However it will wait for a previous write to complete before undertaking a new write operation, so it can be used repeatedly without compromising data integrity.

To read an address then simply use the ReadEEData() function, the parameter is the address. The function returns the value read from the specified address.

Returns:

ReadEEData - Returns value read from EEPROM location.

WriteEEData - Void function - no return.

Examples:

Here is a simple EEPROM programmer utility which can be used with the FED 16F877 development board and the terminal within the C Compiler Debugging window. Press B and follow with another character to read a value from EEPROM, press C and follow with an address and a value to write to a location. For example press C,0,A to write the hex value 0x41 to location 0x30, press B and then 0 to return the value written to that location.

```
#include <pic.h>

const long SERIALRATE_IN=9600;
const int BITTIME_IN=APROCFREQ/SERIALRATE_IN/4;
const long SERIALRATE_OUT=9600;
const int BITTIME_OUT=APROCFREQ/SERIALRATE_OUT/4;

const BYTE SERIALPORT_IN=&PORTC;
const BYTE SERIALBIT_IN=7;
const BYTE SERIALPORT_OUT=&PORTC;
const BYTE SERIALBIT_OUT=6;
unsigned char x;

void main()
{
    unsigned char A,D,x;

    TRISC=~(1<<SERIALBIT_OUT);

    pSerialOut('K');        // Transmit a 'K' to tell system that we're
    here

    while(1)
```

```

{
  x=pSerialIn();
  switch(x)
  {
    case 'B': pSerialOut(ReadEEData(pSerialIn()));
              break;                                // Return from Address
x
    case 'C': A=pSerialIn(); D=pSerialIn();
              WriteEEData(A,D);
    case 'A': pSerialOut('K');
              break;                                // Acknowledge
  }
}
}

```

Graphic LCD Functions

void GLCDInit();

Other functions defined in the library reference

Header file : <GraphicLCD.h>

Library file : <GraphicLCD.c>

These functions are provided in the Graphic LCD library. This library is not included in the compilation by default, but must be added using the File | Libraries menu option. For full details please see the separate manual Graphic LCD Library.pdf supplied in the Libraries sub-directory of the installation CD.

I2C Routines (hllnit)

Software Routines

unsigned char IIRead(unsigned char Definition);

unsigned char IIWrite(unsigned char Data,unsigned char Definition);

void QuickStop();

Hardware Routines

unsigned char hllRead(unsigned char Definition);

unsigned char hllWrite(unsigned char Data,unsigned char Definition);

void hQuickStop();

void hllnit();

Header file : <datalib.h>

Library file : <datalib.c>

These routines are used to drive the I²C bus.

The software versions drive the bus with software control, the hardware versions use the internal PIC hardware. As the bus transfer rate is high the hardware versions do not use interrupts as standard. The hardware and software versions are deliberately very similar and a system written to use the software can be quickly changed to use the internal hardware. The two versions are shown below.

Software versions

IIRRead and IIWrite transfer 8 bits with an optional acknowledgement to the IIC bus.

IIRRead will read 8 bits from the bus and return the read value. The SDA data bit should be set by the program to read before undertaking the function call.

IIWrite will write 8 bits from the parameter Data to the bus. It returns the state of the acknowledgement bit (either 0 or 1). The SDA data bit should be set by the program to drive before undertaking this function call.

QuickStop clocks 8 data bits on the bus and then sends a stop state. It is used to terminate pending operations on the IIC bus.

The parameter Definition defines how the transfer should proceed. It is made up of up to three controls IISTART, IISTOP and IIACK which may be OR'd together. IISTART forces a start bit on the bus before the transfer, IIACK forces the PIC to drive an acknowledgement to the bus (otherwise the acknowledgement bit will be read), and finally IISTOP forces a stop state after the transfer. For example to undertake a read transfer with a START state and to send an acknowledgement bit then the following function call could be used:

```
IIRRead(IISTART | IIACK) ;
```

The routines return driving the data bit if an acknowledgement has been requested, and return reading the data bit if no acknowledgement is requested. Use IINONE if no action is to be used on the bus.

The port and bits used for the IIC port are set up with the following constant integers:

PICIO

_SDA

_SCL

IISstretch

See the examples to see how these are set up. Note that the clock pin and data pins must be set up as outputs.

The only integer not described in the example is IISstretch which sets up the bus speed. It is not compulsory to define IISstretch, if it is not defined (as in the examples) then the bus operates at 400KHz - the high speed rate. To operate the bus at a lower rate then set IISstretch to a multiplier value, then all the bus timing parameters will be multiplied by this value. For example to use a 100KHz bus then all times should be multiplied by four. This is how to set up the II interface for the 100KHz bus:

```
const int IISstretch=4;
```

It may be necessary to increase IISstretch for any bus if the bus wires run for more than a short distance.

Hardware Versions

hIIInit should be called at the start of the program to set up the I2C hardware, the SCL and SDA pins are defined by the PIC pinout. The TRIS bits for the I2C bus bits should be left set to '1' as inputs. The bus drivers are set automatically by the PIC hardware. Note both SCL and SDA pins should be pulled high with 2K resistors.

hIIRead and hIIWrite transfer 8 bits with an optional acknowledgement to the IIC bus. hIIRead will read 8 bits from the bus and return the read value. hIIWrite will write 8 bits from the parameter Data to the bus. It returns the state of the acknowledgement bit (either 0 or 1). hQuickStop undertakes a read operation on the bus and then sends a stop state. It is used to terminate pending operations on the IIC bus.

The parameter Definition defines how the transfer should proceed. It is made up of up to four controls IISTART, IIRESTART, IISTOP and IIAACK which may be OR'd together. IISTART forces a start bit on the bus before the transfer, IIRESTART is similar for a bus which is already in operation, IIAACK forces the PIC to drive an acknowledgement to the bus (otherwise the acknowledgement bit will be read), and finally IISTOP forces a stop state after the transfer. For example to undertake a read transfer of a byte a START state and to send an acknowledgement bit then the following function call could be used:

```
hIIRead(IISTART | IIAACK) ;
```

Use IINONE if no action is to be used on the bus. The routines return driving the data bit if an acknowledgement has been requested, and return reading the data bit if no acknowledgement is requested.

The port and bits used for the IIC port are set up with the following constant integers:

PICIO

_SDA

_SCL

IISstretch

See the examples to see how these are set up. Note that the clock pin and data pins must be set up as outputs.

I2CBusRate sets up the bus speed. It is not compulsory to define I2CBusRate, if it is not defined then the bus operates at 400KHz - the high speed rate. It can be set to 1000 for the top rate. To set up the bus for 100KHz then use :

```
const int I2CBUSRATE=100;
```

It may be necessary to reduce the bus rate for any bus if the bus wires run for more than a short distance.

Returns:

`IIRead()` *hIIRead()* - returns the read value.

`IIWrite()` *hIIWrite()* - returns the state of the acknowledgement bit.

`QuickStop()` *hQuickStop()* - Void function - no return.

`hIIInit()` - Void function - no return.

Examples:

See [EEPROM Programmer](#)

IRDA IR Routines

void IRTx(unsigned char Byte);

void IRRx(void);

extern unsigned char IRRxVal;

Header file : <datalib.h>

Library file : <datalib.c>

These routines are used to drive an IRDA infra-red transceiver according to the IRDA 1.0 standard for infra-red transmission from 9600bps up to 115000 bps (although the FED PIC C implementation is limited by processor clock rate). Only the physical (link) level is supported. IRDA transceivers are recommended as they are optimised for this type of transmission, however discrete circuits may be used. The implementation supports multiple receivers.

The IRDA standard transmits data in the same format as normal asynchronous 8 bit serial data with a low start bit and a high stop bit (10 bits in total). A 0 bit is sent as a pulse with a width of 3/16 of a bit. So at 9600 bps the Zero bit is sent with an IR pulse of approximately 20uS. The idle state is defined as a one bit, so the IR transmitter is normally turned off.

These routines work with a high level logic IRDA transceiver. That is to turn the IR transmitter on the pin of the PIC sends a high bit, and similarly the IRDA transceiver sends a level 1 when an infra-red signal is detected. Both the transmitter and receiver

IRTx sends the supplied byte in IRDA format on the defined pin of the PIC (the description of how to define the pins is presented below).

IRRx receives a byte from one or more pins of the PIC, it will wait until it detects a signal and then will receive the Byte which is not returned, but is placed in the global external variable called IRRxVal. The IRRx function has been designed to be called from a quick interrupt (if required), and uses 3 bytes of global memory. If it is used in a quick interrupt then it must be the first function in the interrupt, it cannot be used in a normal interrupt. To use it in an interrupt then the IRDA receivers should be connected to an interrupt pin of the PIC (for example Port B bit 0, or Port B bits 4 to 7). Now when the start bit is detected an interrupt will be caused and the IRRx routine will detect the rest of the byte (see example below). Note that if connected to one of the Port B change pins then the other pins should be

connected so that they cannot cause interrupts, or so that interrupts are disabled whilst the other pins of Port B are changed.

Note that most IRDA transceivers are optically coupled (either directly or by reflection), and therefore receive the information which is being transmitted, thus interrupts must be disabled during transmission, and approximately 1mS allowed between transmission and reception. As with all software based transmission it is wise to allow time between transmitted bytes for the receiving device to process received information.

The port and bits used for the Infra-Red port are set up with the following constant integers:

IRPORT

IRRATE

IRTXMASK

IRDAMASK

IRINT

IRPORT is the port to which the transmitter and receiver are connected (they must be on the same port). IRRATE is the bit rate to be used which at a 4MHz clock must be no greater than 9600 bps and may be scaled with processor clock frequency (e.g at 16MHz a rate of 38400bps may be used). IRTXMASK is a mask for the bits used for transmitter(s) on the defined port. For example if the transmitter is connected to bit 1 then IRTXMASK will be 2. IRDAMASK is the mask for the bits used for the receiver(s) on the defined port. For example if the receiver is connected to bit 4 then IRTXMASK will be 0x10. IRINT should be defined and set to 1 if the receiver is to be detected in an interrupt routine. See the examples for details of how to set these up.

Returns:

IRTx() - Void function - no return.

IRRx() - Void function - no return, returned byte is placed in variable IRRxVal.

Examples:

The following example shows an infra red receiver which will receive a byte from an IRDA transceiver connected to Port B pin 0. It will then transmit it using asynchronous serial protocols on Port A, bit 3, this is suitable for use with the FED development board, and will transmit received bytes to an attached PC. Note that this example needs at least 1mS between transmitted IR bytes in order to send the information to the serial port.

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

const BYTE IRRATE=9600;           // IR bit rate
const BYTE IRDAMASK=1;           // Mask for received information
const BYTE IRPORT=&PORTB;        // IR port

// Definitions for the serial port

const long SERIALRATE_OUT=9600;
const int BITTIME_OUT=(4000*1000)/SERIALRATE_OUT/4;
```

```

const BYTE SERIALPORT_OUT=&PORTA;
const BYTE SERIALBIT_OUT=3;

BYTE Flag=0;                                // Set when an IR byte is received

void main()
{
    TRISA=~(1<<SERIALBIT_OUT);    // Serial port output
    PORTA=0xff;                    // Initial value of serial output
    INTCON=(1<<GIE) | (1<<INTE);  // Enable interrupt on port B, bit 0

    while(1)                        // Loop forever
    {
        if (Flag) {pSerialOut(IRRxVal); Flag=0;} // Send a received byte
    }

    const int QuickInt=1;           // Define use of quick interrupts
    const int IRINT=1;              // Show use of IR interrupt

    void Interrupt()
    {
        IRRx();                    // Receive byte
        Flag=1;                    // Set a flag
        INTCON&=~(1<<INTF);        // Clear the interrupt
    }
}

```

RC5 IR Routines

void TransmitRC5 (unsigned int Word);

unsigned int GetRC5(void);

void RC5Rx(void);

extern unsigned int RC5Value;

Header file : <datalib.h>

Library file : <datalib.c>

These routines are used to drive a IR LED or to receive from an IR receiver module using the Phillips RC5 code. Normally only the receive or transmit functions will be used on a project unless duplex data transmission is set up.

WIZ-C users may find it easier to use the elements to set up the RC5 system.

This code has 14 bit transmission, the first two bits are always 1, the next bit is a check bit which may be set to 1 or 0, the following 11 bits consist of a 5 bit control address and 6 bits of data. The check bit alternates on each button press on the remote so that if the beam is interrupted the receiving device can check that a new command has not been sent. The control address identifies the device (TV, Video etc) and the command bits the function to support. Normally in RC5 the command repeats at about 100mS intervals. Typically it takes around 22mS to transmit a complete RC5 word.

The FED library functions do not differentiate between command, data and control bits – a 12 bit value is transmitted or received – it is up to the user to split this between check, data and address. The code can transmit at up to about 500 bits per second for data transfer.

These routines work with a low level logic RC5 receiver and high level logic IR LED.. That is to turn the IR LED on the pin of the PIC sends a high bit, however the RC5 receiver sends a level 0 when an infra-red signal is detected and a level 1 (+5v) when no signal is received.

TransmitRC5 sends the supplied 12 bit word in RC5 format on the defined pin of the PIC (the description of how to define the pins is presented below). The check bit is the first bit transmitted followed by bit 10, 9 and so on down to bit 0 of the supplied word.

RC5Rx receives a byte from one or more pins of the PIC, it will wait until it detects a signal and then will receive the Byte which is not returned, but is placed in the global external variable called RC5Value. The RC5Rx function has been designed to be called from a quick or normal interrupt (if required), and uses 4 bytes of global memory in addition to the variable RC5Value. To use it in an interrupt then the IRDA receivers should be connected to an interrupt pin of the PIC (for example Port B bit 0, or Port B bits 4 to 7, or GPIO bit 2). The interrupts should be set to trigger on a falling edge. Now when the start bit is detected an interrupt will be caused and the RC5Rx routine will detect the rest of the byte (see example below). Note that if connected to one of the Port B change pins then the other pins should be connected so that they cannot cause interrupts, or so that interrupts are disabled whilst the other pins of Port B are changed.

GetRC5 is used when interrupts are not required – the routine waits for the next word to be received and returns it as well as loading it into the variable RC5Value.

Note that most receivers suffer from occasional noise when no signal is present, and also at distance the signal may become corrupted. If the signal is not recognised, or is not in RC5 format the GetRC5() and RC5Rx() functions return the value 0xffff in the RC5Value variable.

The port and bits used for the Infra-Red port are set up with the following constant integers:

IRFREQ (used when TransmitRC5, GetRC5 or RC5Rx are used)

IRRXPOR (used when GetRC5 or RC5Rx are used)

IRRXBIT (used when GetRC5 or RC5Rx are used)

IRPOR (used when TransmitRC5 is used)

IRBIT (used when TransmitRC5 is used)

IRRXPOR is the port to which the receiver is connected (they must be on the same port). IRPOR is the port to which the IR LED for transmission is connected. IRRXBIT and IRBIT are the bit numbers of the receiver and IR LED respectively. IRRATE is the carrier frequency to be used which must be matched to the receiver centre frequency. Typcially 36000, See the examples for details of how to set these up.

Returns:

void TransmitRC5 (unsigned int Word); Void function - no return.

void RC5Rx (unsigned int Word);
global

Void function returned value is placed in the
variable RC5Value.

unsigned int GetRC5(void);

Returns the received word

IRTx()

IRRx() - Void function - no return, returned byte is placed in variable IRRxVal.

Examples:

The following example shows an IR receiver programme using interrupts – the receiver is connected to PORT B bit 0.

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

const int IRFREQ=38000;
const int IRRXPORT=&PORTB;
const int IRRXBIT=0;

int x;

void main()
{
    bINTEDG=0;           // Interrupt on falling edge of RB0
    bINTE=1;             // Enable Edge triggered Interrupt
    bGIE=1;
    RC5Value=0xffff;     // Flag for received value

    while(1)
    {
        if (RC5Value!=0xffff) {x=RC5Value; RC5Value=0xffff;}
    }
}

const int QuickInt=1;    // Show we must use Quick Interrupts

void Interrupt()
{
    if (bINTF)           // Test interrupt flag
    {
        bINTF=0;        // Clear interrupt flag
        RC5Rx();         // Call reception routine
    }
}
```

The following example shows an IR transmission programme for the 16F675, in this case the transmitter is connected to pin GPIO bit 0 . The programme transmits the code 0x7f on reset.

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

const int IRFREQ=38000;
const int IRPORT=&PORTA;
const int IRBIT=0;

void main()
{
    TransmitRC5('~');
}
```

}

The following stimulus file is an example of transmission of a code – in this 0x3F1 into pin 0 of port B at an IR carrier rate of 38KHz, it is useful for checking operation, copy into a single column of a STI file.

	PORTB: 0=0	36842u
0	28421u	PORTB: 0=0
20000u	PORTB: 0=1	37684u
PORTB: 0=1	29263u	PORTB: 0=1
20842u	PORTB: 0=0	38526u
PORTB: 0=0	30105u	PORTB: 0=0
21684u	PORTB: 0=1	39368u
PORTB: 0=1	30947u	PORTB: 0=1
22526u	PORTB: 0=0	40210u
PORTB: 0=0	31789u	PORTB: 0=0
23368u	PORTB: 0=1	41052u
PORTB: 0=0	32631u	PORTB: 0=1
24210u	PORTB: 0=0	41894u
PORTB: 0=1	33473u	PORTB: 0=1
25052u	PORTB: 0=1	42736u
PORTB: 0=0	34315u	PORTB: 0=0
25894u	PORTB: 0=0	43578u
PORTB: 0=1	35157u	PORTB: 0=1
26736u	PORTB: 0=1	
PORTB: 0=1	36000u	
27578u	PORTB: 0=0	

FED provide an Excel spreadsheet which can be used to generate the stimulus file for any RC5 sequence – it is on the CD and is called RC5.xls.

Interrupt Driven Serial Port

void AddTx(unsigned char TxChar); void AddTx2(unsigned char TxChar);

unsigned char GetRxSize(void); unsigned char GetRxSize2(void);

unsigned char GetTxSize(void); unsigned char GetTxSize2(void);

unsigned char WaitRx(void); unsigned char WaitRx2(void);

void SerIntHandler(void); void SerIntHandler2(void);

void SerIntInit(void); void SerIntInit2(void);

Header file : <datalib.h>

Library file : <datalib.c>

These routines are used to drive the serial interface hardware of all supported devices with serial interface. These transmit and receive data on pins C6 and C7 for the majority of devices and other pins for smaller (and bigger) PIC's. The user defines buffers for transmission and reception and the device will operate on interrupts receiving and transmitting bytes in the background. XON/XOFF protocols may be used to flow control the link. The example below shows a simple program which will loop received characters back to the transmitter but with a 5 mS delay inserted so that flow control must be utilised.

For all of the routines the same function may be called with a 2 appended to drive the USART 2 hardware for those devices which have it.

The values and variables used for the Serial Routines are set up with the following constant integers and variables:

```
TXBUFSZ;
```

```
RXBUFSZ;
```

```
SERINTRATE;
```

```
USEXON;
```

```
unsigned char TxTab[TXBUFSZ];
```

```
unsigned char RxTab[RXBUFSZ];
```

```
TXBUFSZ2;
```

```
RXBUFSZ2;
```

```
SERINTRATE2;
```

```
USEXON2;
```

```
unsigned char TxTab[TXBUFSZ2];
```

```
unsigned char RxTab[RXBUFSZ2];
```

TXBUFSZ and RXBUFSZ are the size of the transmit and receive buffers. These must be a power of 2, and (as described below) FED recommend a size of 32 for the receive buffer if the system is communicating with a standard PC, typically the transmit buffer (TXBUFSZ) may be 8 or 32 bytes. SERINTRATE should be set to the required serial bit rate, e.g. 9600. USEXON should be set to 0 or 1, if set to 1 then XON/XOFF signalling will be used, otherwise no handshaking is used. TxTab and RxTab should be defined at the top of the program exactly as shown.

The system is initialised with SerIntInit(). This void function sets up the serial communication registers and defines the transmit and receive bits as outputs and inputs, however it is recommended that users manually set the TRIS bits for devices as well. SerIntInit() should be called at the beginning of the program. Please note this function leaves interrupts enabled.

The function SerIntHandler() must be called from within the Interrupt routine. This allows the use of Quick Interrupts. See the example which shows how this may be achieved.

Now the AddTx(Value) function will add the byte Value to the transmit buffer, if there is no room then the function will wait until room is available (when a byte is next transmitted by the interrupt and hardware). The WaitRx() function waits until a character is available and

returns it. The GetTxSize() and GetRxSize() functions return the size of the transmit and receive buffers respectively. Therefore if GetRxSize() returns a non-zero value there is a byte waiting in the buffer, similarly if GetTxSize() is the same size as the defined transmit buffer size then there is no room in the transmit buffer.

Please note the following if XON/XOFF signalling is in use. An XOFF character will be sent when the receive buffer is 1/2 full, and an XON character is sent when the buffer is 1/8 full again. If communicating to a PC the serial port on the PC usually has a buffer of 8 characters which will be emptied regardless of received XOFF characters, therefore to avoid missing characters it is essential that the RXBUFSIZE value is set to 32 bytes. Also when the system is initialised an XON character is sent immediately by SerIntInit().

With XON/XOFF signalling if an XON, XOFF or ESCAPE (0x11, 0x13, 0x1b) character is transmitted by AddTx(), then the character will be sent as an ESCAPE character (0x1b), followed by the actual character with bit 7 set. Thus if AddTx(0x11) is called then characters 0x1b and 0x91 are transmitted. Similarly when an escape (0x1b) is received then the following character with bit 7 reset is then added to the receive buffer, the 0x1b is not added. This allows the complete character set from 0 to 0xff to be transmitted and received.

In all cases if the receive buffer is full and a new character is received, then the oldest character in the buffer is overwritten.

Returns:

AddTx(); Void function, no return.

GetRxSize(); Returns the number of bytes in the receive buffer.

GetTxSize(); Returns the number of bytes in the transmit buffer.

WaitRx(); Returns the next byte from the receive buffer.

SerIntHandler(); Void function, no return.

SerIntInit(); Void function, no return.

Examples:

The following example shows an application which will loop back every received character, but inserts a 5mS delay before doing so, thus XON/XOFF handshaking is essential. This application is in the projects directory of the C compiler, under the Serial Interrupts sub-directory. The bit rate is 9600 and the application is written for the 16F877.

```
//
// A test program for serial interrupts.
//

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

const int TXBUFSZ=32;
const int RXBUFSZ=32;
const int SERINTRATE=9600;
const int USEXON=1;
unsigned char TxTab[TXBUFSZ];
```

```

unsigned char RxTab[RXBUFSZ];
unsigned char x,RxSz,TxSz;

void main()
{
    SerIntInit(); // Initialise serial interrupts
    AddTx('K');   // Transmit a 'K' to tell system that we're here

    while(1)
    {
        x=WaitRx();
        Wait(5); // Test XON/XOFF
        AddTx(x);
        RxSz=GetRxSize(); // Information only
        TxSz=GetTxSize(); // Information only
    }
}

const int QuickInt=1; // Quick interrupts

void Interrupt() // Interrupt handler
{
    SerIntHandler();
}

```

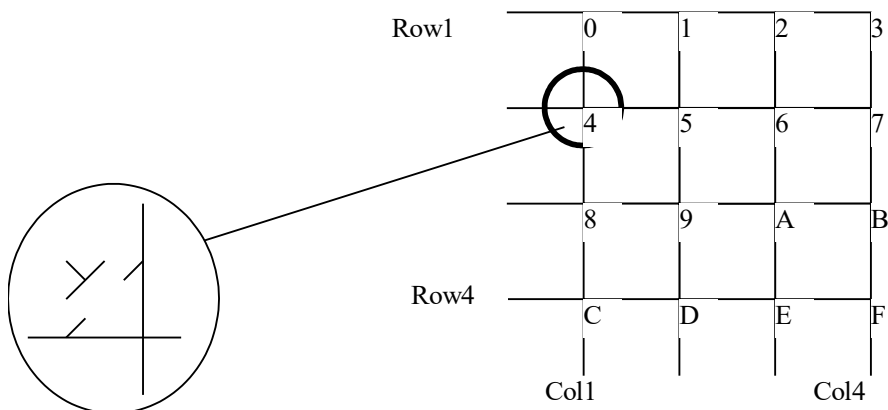
Hex Keypad

unsigned char KeyScan();

Header file : <displays.h>

Library file : <displays.c>

This function is provided to scan a hex keypad with 4 rows and 4 columns. The rows are driven and the columns are read and should be pulled up with resistors :



The rows are outputs from the PIC, the columns are inputs, and may be shared with other inputs, the inputs must be pulled up with resistors of 2K or more. The outputs may in similar fashion be shared with other outputs, the KeyScan routine sets the outputs to drive, but if the value KeyPadRelease is set to 1 then they will be set back to inputs after the keypad is scanned. There is no debounce facility, this may be achieved within the main program.

The keypad is defined with 9 constant integers which indicate the port and bit number for each row and column and define keypad release, see the example program below:

```
const int Row1Port    Port number for row 1

const int Row2Port    Port number for row 2

const int Row3Port    Port number for row 3

const int Row4Port    Port number for row 4

const int Col1Port    Port number for Column 1

const int Col2Port    Port number for Column 2

const int Col3Port    Port number for Column 3

const int Col4Port    Port number for Column 4

const int Row1Bit     Bit number for Row 1

const int Row2Bit     Bit number for Row 2

const int Row3Bit     Bit number for Row 3

const int Row4Bit     Bit number for Row 4

const int Col1Bit     Bit number for Column 1

const int Col2Bit     Bit number for Column 2

const int Col3Bit     Bit number for Column 3

const int Col4Bit     Bit number for Column 4

const int KeyPadRelease    Set to 0 if rows are to remain driving after keyscan, or set to
1 if rows are to return to inputs.
```

Returns:

The function returns 0xff if no key is pressed, and a key number from 0 to 15 if a key is detected.

Example:

The following example for the 16F877 on the FED PIC demonstration board sends any key presses detected to the serial port which can be read on an attached PC. A new value is only sent if it is different to the last value, and if a value has been detected.

```
//
// Hex keypad scanner - read key and transmit as hex value if it
// changes
//
#include <pic.h>
#include <Displays.h>
```

```

#include <Datalib.h>
const int Row1Port=&PORTD;           // Set up keypad
const int Row2Port=&PORTD;
const int Row3Port=&PORTD;
const int Row4Port=&PORTD;
const int Col1Port=&PORTD;
const int Col2Port=&PORTD;
const int Col3Port=&PORTD;
const int Col4Port=&PORTD;
const int Row1Bit=0;
const int Row2Bit=1;
const int Row3Bit=2;
const int Row4Bit=3;
const int Col1Bit=4;
const int Col2Bit=5;
const int Col3Bit=6;
const int Col4Bit=7;
const int KeyPadRelease=0;           // Leave driving when not in
use

const int TXBUFSZ=8;
const int RXBUFSZ=32;
const int SERINTRATE=9600;
const int USEXON=0;
unsigned char TxTab[TXBUFSZ];
unsigned char RxTab[RXBUFSZ];

void main()
{
    char ox=-1,nx;

    SerIntInit();                     // Initialise serial interrupts
    AddTx('K');                       // Transmit a 'K' to tell system that we're
    here

    while(1)
    {
        nx=KeyScan();
        if (nx==ox || nx==0xff) {ox=nx; continue;}
        ox=nx;
        nx+='0';
        if (nx>'9') nx+='A'-'9'-1; //Hex display
        AddTx(nx);
    }
}

const int QuickInt=1;

void Interrupt()
{
    SerIntHandler();
}

```

isFunctions

unsigned char isalpha(char c);

unsigned char isalnum(char c);

unsigned char isascii(char c);

unsigned char iscntrl(char c);

unsigned char isdigit(char c);

unsigned char islower(char c);

unsigned char isprint(char c);

unsigned char ispunct(char c);

unsigned char isspace(char c);

unsigned char isupper(char c);

unsigned char isxdigit(char c);

char toascii(char c);

char tolower(char c);

char toupper(char c);

Header file : <ctype.h>

Library file : <ctype.c> <ctype16.c>

These functions are provided in the ctype libraries. These libraries are not included in the compilation by default, but must be added using the File | Libraries menu option. For full details please see the separate manual [LibraryExtensions.pdf](#).

LCD

```

void LCD(unsigned char Data);           // Send character to display
void LCDClear();                       // Clear display return print position to 0,0
void LCDClearLine();                  // Clear all remaining characters on current line
void LCDFree(void);                   // Wait until LCD is free before returning
void LCDCommand(unsigned char Cmd);    // Send the command to the display (RS is set to 0)
void LCDPrintAt(unsigned char x,unsigned char y); // Set print to line y, character x (top left is 0,0)
void LCDOnOff(unsigned char display,unsigned char cursor,unsigned char blink); // Turn
Module on/off
void LCDShift(unsigned char cursor,unsigned char right);
char LCDBufferClear();                // Return true if buffer is clear
void LCDClearBuf();                   // Clear screen and buffer
void EnableLCD(unsigned char Enable); // Enable the LCD display by setting LCDPresent flag
void LCDHex(unsigned char x);         // Print value in Hex
void LCDInit(unsigned char nLines);   // Initialise LCD (nLines is 1 or 2, if 4 line, then nlines is 2)
void LCDPrintAt(unsigned char x,unsigned char y); // Print at Character,Line
void LCDPrintChar(char x);            // Print character as number on LCD
void LCDPrintuChar(unsigned char x);  // Print unsigned character as number on LCD
void LCDPrintBYTE(unsigned char x);   // Print unsigned character as number on LCD
void LCDPrintBYTELZ(unsigned char x); // Print character as number with leading zero
void LCDPrintInt(int x);              // print an integer as number on the LCD
void LCDString(unsigned char *String); // Print a string on the LCD

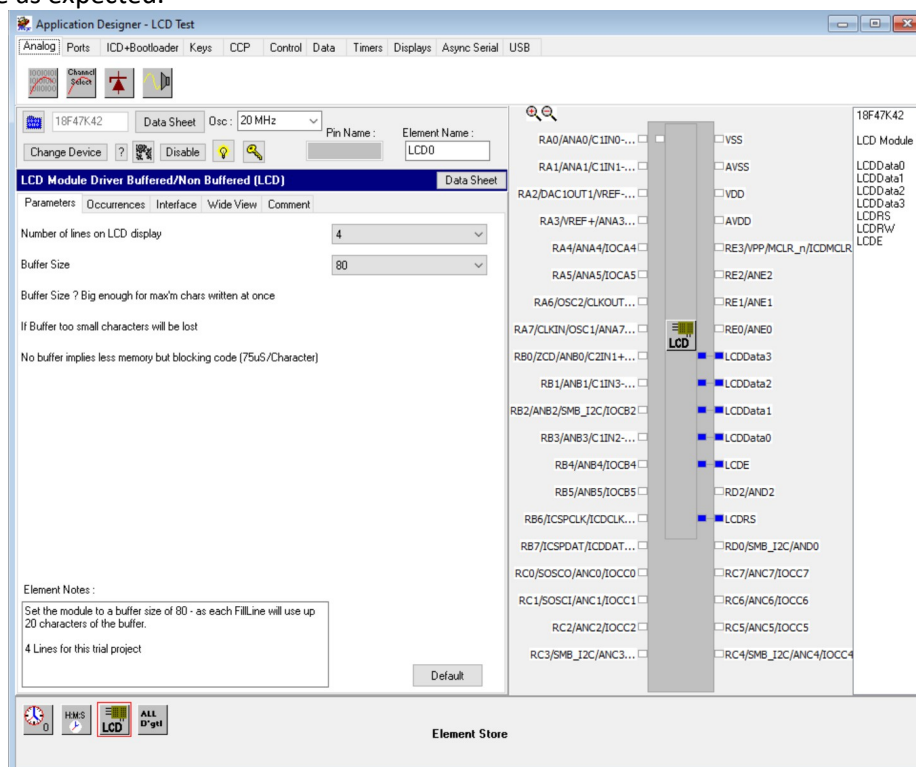
```

unsigned char LCDPresent; // Flag set to non-zero if display is detected (needs RW pin connected)

Header file : <displays.h>

Library file : <displays.c>

These functions are provided to write data to a connected LCD display based on the Hitachi chip set (by far the most common chip used in LCD modules). The library should only be used through the Application Designer and this allows the module to be connected and the number of lines set and whether a buffer should be used. The library is set up to use 2x16 or 4x20 displays which are almost universally the only displays available. Other sizes are rare and will operate with the library however the printat() and clearline() functions may not operate as expected.



The LCD buffer is set up to allow rapid non-blocking printing to the display. The buffer stores characters and commands and then outputs them to the display every milli-second. The buffer size is set in the application designer and should be sized to be large enough to store all the characters which will be written at once. So a 2x16 LCD display might have a buffer of 40 characters noting that some commands – such as clearing the display – take up 8 characters in the buffer, sending a single character only takes one byte in the buffer. A 4x20 line display will be best used with an 80 or 128 character buffer – again smaller buffers can be used if less characters are written together. If the buffer is too small characters will be lost.

If the buffer is used then the application wizard sets up a call to print the next character from the buffer once every milli-Second.

If the buffer is not used then less memory is used for both the RAM and program memory however the code is blocking and every character will hold up the main programme for 50uS, the clear command blocks for 2mS.

The use of the buffer is transparent to the library commands – for example the LCD(char c) function sends a character to the display, if a buffer is used then it is written to the buffer and the function returns immediately, if a buffer is not used then it will be written to the display directly after which there will be a 50uS delay before returning.

The LCDString function will take the '\n' character to start printing on the next line, and the '\r' character to start printing at the beginning of the current line.

The RW connection is not required but may be connected to marginally improve performance, it is also useful if the display is optional. If RW is connected then if no display is attached the LCDPresent variable will be set to 0, or to a non-zero value if a display is detected.

LCD Module	Port number	Note
RS	e.g. BIT1	Any digital Port/Bit
R/W	e.g. BIT2	Optional, if not used then connect the R/W pin of the display to ground. Only really useful to detect if a display is present in which case the LCDPresent variable is set to a non-zero value if module is detected and all functions will return quickly without executing code.
E	e.g. BIT3	Any digital Port bit
D4	BIT4	Bits 4 to 7 may be connected to bits 4 to 7 of any port, but must all be connected to the same port. They may be shared with other functions, the port is left as tristate input after functions return.
D5	BIT5	
D6	BIT6	
D7	BIT7	

Bits D0-D3 of the display may be left floating.

Returns:

See functions

Examples:

A full description and example of the use of an LCD Display is presented in the examples section.

```
LCDClear();           // Clear display
LCD('A');             // Print character A
LCDPrintAt(0,1);      // Set print to start of line 1
LCDString("Welcome"); // Print string on display
```

Maths Routines***float arcsin(float v);******float arccos(float v);******float arctan(float v);******float cos(float v); etc.******float exp(float y);******float exponent(float y);******float fabs(float v);******char * fPrtString(char *String,float v);******float log(float y);******float log10(float y);******float mantissa(float y);******float pow(float x, float y);******float pow10(float y);******float PowerSeries(float v,float *Coefficients,char n,char indices);******float sin(float v);******float sqrt(float v);******float tan(float v);******LN10******e******PI******Header file : <maths.h>******Library file : <maths.c>***

These routines calculate various transcendental and power functions for floating point numbers. Note that these functions are all heavily processor intensive, and typically take between 100uS and 10mS to complete with a 20MHz clock. All routines have been optimised for speed over the entire numeric range at the expense (on occasion) of small number performance.

There is no range or error checking with any of these functions to ensure maximum speed. It is up to the user to supply parameters in range, or provide range checking externally to the

functions. The accuracy of the function is as good as the numeric type barring rounding errors in the power functions which accumulate to typically one significant figure.

All trigonometric functions operate in radians.

Operations:

arccos Returns \cos^{-1} of the supplied parameter. The parameter *must* be greater than -1 and less than $+1$.

arcsin Returns \sin^{-1} of the supplied parameter. The parameter *must* be greater than -1 and less than $+1$.

arctan Returns \tan^{-1} of the supplied parameter. The parameter *must* be greater than -1 and less than $+1$.

cos Returns the cosine value of the parameter v .

LN2 A macro. Defined as 0.69314718

e A macro. Defined as 2.7182818.

exp Returns e to the power y . (e^y)

exponent Returns the exponent value of the floating point number. For example the decimal number 5 is represented as 1.01×2^2 . This function will return 2 in this example. Used in library functions.

fabs Returns the absolute value of the parameter v .

fPrtString Prints the number v in floating point form to the string "String".

LN10 A macro. Defines as natural log 10, 2.3025851.

log Returns the natural logarithm of y ($\ln y$).

log10 Returns the logarithm of y to the base 10

LOG2_10 A macro. Defined as 3.32193, $\log_2(10)$

mantissa Returns the mantissa value of the floating point number. For example the decimal number 5 is represented as 1.01×2^2 . This function will return 1.25 in this example. Used in library functions.

PI A Macro, defined as 3.14145962.

pow Returns x to the power y . (x^y)

pow10 Returns 10 to the y (10^y).

PowerSeries This function is used by library routines and calculates the sum of a power series. v is the value to which the series is applied. Coefficients is a pointer to an array of coefficients to be multiplied by each term in the series in turn. n is the number of terms to

apply (the series will terminate early however if it converges). Finally indices is 0 if the series applies to each power of v, 1 if it applies only to odd powers and 2 for even powers. Returns the sum of the power series.

sin Returns the cosine value of the parameter v.

sqrt Returns the square root of v.

tan Returns the tan value of the parameter v.

Examples:

Here is a test program which uses most of the Maths Routines and prints the result of the calculations to a string - ws.

```
#include <maths.h>

//
// Note won't work on 2K devices with no optimisation as stack
// use is too heavy
//

float y,cs;
char ws[16];

void main()
{
    y=sin(PI/4);          cs+=y; // 7.071068e-1
    y=exp(1.5);           cs+=y; // 4.481688e0
    y=mantissa(5);        cs+=y; // 1.25e0
    y=exponent(5);        cs+=y; // 2.00e0
    y=log(1024.000);      cs+=y; // 6.931473e0
    y=cos(-.9);           cs+=y; // 6.216100e-1
    y=tan(-1.3);          cs+=y; // -3.602102e0
    y=log10(.7);          cs+=y; // -1.549019e-01
    y=pow10(0.67);        cs+=y; // 4.677328e+00
    y=2*fabs(-7)-2;       cs+=y; // 1.2e01
    y=sqrt(169);          cs+=y; // 1.3e-1

    fPrtString(ws,cs);

endit:
    while(1);
}
```

mem functions

```
void * memccpy(void * dest,void * src,int c,int n);
```

```
void * memchr(void *dest, int c,int n);
```

```
int memcmp(void *s1, void *s2,int n);
```

```
void *memset(void *s, int c,int n);
```

```
void * memcpy(void * dest,void * src,char n);
```

```
void * memcpy1(void * dest,void * src,int n);
```

```
void * memmove(void * dest,void * src,char n);
```

Header file : <mem.h>

Library file : <mem.c>

These functions are provided in the mem.c library. This library is not included in the compilation by default, but must be added using the File | Libraries menu option. For full details please see the separate manual LibraryExtensions.pdf.

printf Functions

```
void fnprintf(void (*out)(char x),char *str,...);
```

```
void fnprintfsm(void (*out)(char x),char *str,...);
```

```
void printf(char *str,...);
```

```
void printfsm(char *str,...);
```

```
void sprintf(char *Dest,char *str,...);
```

```
void sprintfsm(char *Dest,char *str,...);
```

Header file : <strings.h>

Library file : <strings.c>

These routines are used to print numbers, characters and strings in a user controlled formatted form to a strings or output devices. There are two versions of the functions, the normal and the compact version which has the same name with the letters sm appended. The compact version is much less capable than the normal version, but is about half the size and is more suitable for the smaller PIC's.

The printf function prints characters using the pSerialOut function. Therefore the serial interface must be set up before using printf. This is described in the manual entry for [SerialOut](#). Also to tell the compiler that the printf function is using SerialOut it is also necessary to add the following line to the source code:

```
#callfunction pSerialOut
```

The `fprintf` function prints characters to a user supplied function. This function is set up to output one character to an output device, built in functions (such as the `LCDc` function) may be used, or the user may write a function – examples of both are shown below. The `sprintf` function prints characters to a supplied string.

The name of the function call for `fprintf` is the first parameter of the function. For example the following function call prints the time held in three variables, hours, minutes and seconds to an attached LCD. (More extensive examples are shown later on).

```
fprintf(LCDc, "% 2d:%02d:%02d", hours, minutes, seconds) ;
```

The pointer to the string which is to hold the output is the first parameter of the `sprintf` function. For example the following function call prints the time held in three variables, hours, minutes and seconds to a string. (More extensive examples are shown later on).

```
char ws[16] ;
sprintf(ws, "% 2d:%02d:%02d", hours, minutes, seconds) ;
```

Definition string for `fprintf` and `sprintf`.

The definition string (the `str` parameter) defines the string to be printed, and the format of any parameters supplied for printing. Each character in `str` is printed one after another. Whenever a `%` sign is encountered it specifies a parameter is to be printed.

The parameters are supplied after `str` in order that the definition string shows them. Note that characters are converted to 16 bit values before being handled.

For the definition string (`str`) the `%` sign is followed by optional modifiers and then a specifier for the output:

```
%[ +-][[0]n][l][dcuxXs]
```

If a `+` is supplied then a number will be printed with a leading `+` or `-`.

If a `-` is supplied for numbers it has no effect. For a string it causes the string to be right justified with spaces padding the result if the width (`n`) is greater than the string width.

`n` is the width to be printed. If the number or string is less than `n` characters then padding characters are printed to fill the width up to `n` characters. If the width starts with a `0` then numbers are printed with leading zeroes to fill the width, this has no effect for strings.

`l` (lower case `L`) causes the function that the parameter is long.

The parameter specifiers (one of `d`, `c`, `u`, `x`, `X` or `s`) specify how the parameter is to be printed.

`d` causes the parameter to be printed as an integer (2 bytes, or 4 if the `l` modifier is used).

`c` causes the parameter to be printed as a single character

`u` causes the parameter to be printed as a single character

`x` causes the parameter to be printed as a hex value (lower case `A` to `F`)

X causes the parameter to be printed as a hex value (upper case A to F)

s causes the parameter to be printed as a string

To print a % sign use %%.

The examples below show how various definition strings appear:

```
char *s="ABC";
char c='A';
int x=-1;
long y=0x123456;
char ws[32];

sprintf(ws,"%d",x);           // ws holds "-1"
sprintf(ws,"%u",x);           // ws holds "65535"
sprintf(ws,"%5d",x);          // ws holds "  -1"
sprintf(ws,"%05d",x);         // ws holds "-0001"
sprintf(ws,"%x",x);           // ws holds "ffff"
sprintf(ws,"%X",x);           // ws holds "FFFF"
sprintf(ws,"%04X",c);          // ws holds "0041"
sprintf(ws,"%c",c);           // ws holds "A"
sprintf(ws,"%lx",y);           // ws holds "123456"
sprintf(ws,"%ld",y);           // ws holds "1193046"
sprintf(ws,"%s",s);            // ws holds "ABC"
sprintf(ws,"%5s",s);           // ws holds "ABC  "
sprintf(ws,"%-5s",s);          // ws holds "  ABC"
sprintf(ws,"%c,%d,%l",c,x,y); // ws holds "A,-1,1193046"
```

Note – if the last character of the definition string is a % sign then the string printed will not be terminated with a zero character. This is important for the fprintf function which outputs to a device. If the device cannot handle a zero (such as an LCD display) then the last character should be %.

Definition string for fprintfsm and printfsm.

Note that these functions can only handle basic outputs. Definition strings can only hold %d, %u, %c and %s. All other modifiers – width, sign, justification and long values – cannot be used with these functions.

Use of normal and compact versions.

It is strongly recommended that the normal version or the compact version is used throughout without mixing calls. Thus if your program needs sprintf at any point then use sprintf throughout rather than printfsm.

Special note for use of functions with fprintf.

Owing to a quirk in the method used to compile library functions it is necessary to declare the call supplied to fprintf in the main file. This is achieved by the following pragma - .

```
#callfunction FunctionName
```

For example:

```
#callfunction LCDc
```

This only needs to be declared once in the source, and then only if the function call is not used directly.

Returns:

fnprintf – void function, no return.

fnprintfsm – void function, no return.

sprintfsm – void function, no return.

sprintfsm – void function, no return.

Examples:

Here is a program to print “Hello World” using printf and the serial interface using PORT B bits 0 (for transmission) and bit 1 (for reception).

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

const long SERIALRATE=9600;

const int BITTIME_IN=APROCFREQ/SERIALRATE/4;
const int BITTIME_OUT=APROCFREQ/SERIALRATE/4;
const BYTE SERIALPORT_OUT=&PORTB;
const BYTE SERIALPORT_IN=&PORTB;
const BYTE SERIALBIT_OUT=0;
const BYTE SERIALBIT_IN=1;
#pragma callfunction pSerialOut

void main()
{
    bRB0=1; // Initialise output to level 1
    bTRB0=0;
    printf("%s %d\n%", "Hello World", 1);

    endit:
    while(1);
}
```

Here is a complete test program which prints the contents of memory from address 0x20 to address 0x7f on an attached terminal. It uses the internal interrupt driven serial port library. Note the use of “#callfunction AddTx” which is used to tell the compiler to link in the AddTx function. Note also that the last character of the definition string in the fnprintf function is a % character which suppresses the final terminating zero normally printed by fnprintf.

```
//
// Dump contents of memory to terminal
//
#include <datalib.h>
#include <delays.h>
#include <strings.h>
#include <pic.h>

const int TXBUFSZ=32;
const int RXBUFSZ=32;
const int SERINTRATE=9600;
const int USEXON=1;
extern unsigned char TxTab[TXBUFSZ];
extern unsigned char RxTab[RXBUFSZ];
#pragma locate TxTab 0xa0
#pragma locate RxTab 0xc0
unsigned char x, RxSz, TxSz;
```

```
#callfunction AddTx

void main()
{
    char *i;

    SerIntInit(); // Initialise serial interrupts
    for(i=(char *)0x20; i<=(char *)0x7f; i++)
    {
        fprintf(AddTx,"%04X - %02X\n",i,*i);
    }
}

const int QuickInt=1; // Quick interrupts

void Interrupt() // Interrupt handler
{
    SerIntHandler();
}
```

Here is a test program which prints characters to Port B. Each character is output to Port B and then a strobe line (Port A, bit 0) is asserted high and then low again to “clock out” the character. This program uses the compact version (fprintfsm). Note that the function which “prints” to port B is declared as void return with a single character parameter, note also that “#asmfunc PrintB” is used – this prevents the optimiser from optimising the PrintB function which is essential for the correct operation of fprintf and fprintfsm.

```
//
// Dump a string to port B, strobed on port A0
//
#include <strings.h>
#include <pic.h>

void PrintB(char x);
#asmfunc PrintB(char x)

void main()
{
    ADCON1=7; // A to D converters set to digital
    PORTB=0;
    TRISB=0; // Port B to outputs
    PA.B0=0; // Strobe line low (Port A, bit 0)
    TRISA=~1; // Port A, bit 0 to output

    fprintfsm(PrintB,"A test string - Number %d, Character
%c",123,'F');

    while(1);
}

void PrintB(char x)
{
    PORTB=x;
    PA.B0=1; // Strobe clock line
    PA.B0=0;
}
```

Here is a complete program to print a floating point number using sprintfsm.

```
#include <maths.h>
#include <strings.h>
char fs[16];
char out[32];
```

```
float f=3.141;

void main()
{
    fPrtString(fs,f);
    sprintfsm(out,"Floating Number %s",fs);
endit:
    while(1);
}
```

Random Numbers

void srand(int seed);

int rand(void);

Header file : <maths.h>

Library file : <maths.c>

These routines are used to generate pseudo random numbers.

srand seeds the random number generator, it is recommended that a value such as timer 0 is used to initialise this to avoid the same value being generated on each occasion. rand returns a 16 bit signed number which will appear to the user to be random.

Returns:

srand – no return.

rand returns the random number.

SerialIn

unsigned char SerialIn(char *Port,int BitNumber);

unsigned char pSerialIn(void);

Header file : <datalib.h>

Library file : <datalib.c>

These routines are used to detect asynchronous serial data at an input PIN of the PIC. The routines wait until the falling edge of the start bit of the serial data byte. The entire byte is then received and returned as an unsigned character.

The first version of the routine allows for the user to define the port and the bit to be used to detect the received data. The second version sets up the port and the bit with constants which are defined in the C source file, and therefore this routine always operates on the same port, and the same bit. The second version is more useful when there is only one serial port connected to the PIC, it also uses less space and is slightly faster.

For both versions a global constant must be defined which is the serial bit rate to be used. This is called **BITTIME_IN**. For the **pSerialIn** routine two constants must be defined :

SERIALPORT_IN and **SERIALBIT_IN**, the port defined by these constants will always be used by **pSerialIn** to receive data.

Returns:

The received serial data byte. Note that all other processing is suspended until a start bit is detected and the byte is received.

Examples:

```
//
// Example 1 - SerialIn()
//

//
// Receive a byte at 9600bps on Port B, bit 1 with 4MHz clock
//
#include <pic.h>
#include <datalib.h>

const long SERIALRATE_IN=9600;
const int BITTIME_IN=(4000*1000)/SERIALRATE_IN/4;

unsigned char x;

void main()
{
    x=SerialIn(&PORTB,1);
    while(1);
}

//
// Example 2 - pSerialIn()
//

//
// Receive a byte at 9600bps on Port B, bit 1 with 4MHz clock
//
#include <pic.h>
#include <datalib.h>

const long SERIALRATE_IN=9600;
const int BITTIME_IN=(4000*1000)/SERIALRATE_IN/4;
const BYTE SERIALPORT_IN=6;
const BYTE SERIALBIT_IN=1;

unsigned char x;

void main()
{
    x=SerialIn();
    while(1);
}
```


SerialOut

unsigned char SerialOut(char *Port,unsigned char Bit,unsigned char Transmit);

unsigned char pSerialOut(unsigned char Transmit);

Header file : <datalib.h>

Library file : <datalib.c>

These routines are used to transmit asynchronous serial data at an output PIN of the PIC. The routines expect the output pin to have been defined as an output using one of the Tri-State registers, and set high in the idle state. A zero start bit is transmitted first followed by 8 data bits (LSB first), and finally a high stop bit - 10 bits in total.

The first version of the routine allows for the user to define the port and the bit to be used to transmit the data. The second version sets up the port and the bit with constants which are defined in the C source file, and therefore this routine always operates on the same port, and the same bit. The second version is more useful when there is only one serial port connected to the PIC, it also uses less space and is slightly faster.

For both versions a global constant must be defined which is the serial bit rate to be used. This is called **BITTIME_OUT**. For the **pSerialIn** routine two constants must be defined : **SERIALPORT_OUT** and **SERIALBIT_OUT**, the port defined by these constants will always be used by **pSerialOut** to transmit data.

Returns:

Void function - no return.

Examples:

```
//
// Example 1 - SerialOut()
//

//
// Transmit byte 0x55 at 9600bps on Port B, bit 1 with 4MHz clock
//
#include <pic.h>
#include <datalib.h>

const long SERIALRATE_OUT=9600;
const int BITTIME_OUT=(4000*1000)/SERIALRATE_OUT/4;

void main()
{
    SerialOut(0x55);
    while(1);
}

//
// Example 2 - pSerialOut()
//

//
// Transmit byte 0x55 at 9600bps on Port B, bit 2 with 4MHz clock
```

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

const long SERIALRATE_OUT=9600;
const int BITTIME_OUT=(4000*1000)/SERIALRATE_OUT/4;
const BYTE SERIALPORT_OUT=6;
const BYTE SERIALBIT_OUT=2;

unsigned char x;

void main()
{
    pSerialOut(0x55);
    while(1);
}
```

stdio functions

char getc();

char getchar();

void gets(char *s);

char fgetc(FILE fi);

char fgets(char *s,FILE fi);

char fprintf(FILE fi,char *s,...);

void fputc(char c,FILE fo);

void fputs(char *s,FILE fo);

unsigned char kbhit();

char printf(FILE fi,char *s,...); // See printf functions

void putc(char c);

void putchar(char c);

void puts(char *s);

Header file : <stdio.h>

Library file : <stdio.c>

These functions are provided in the stdio.c library. This library is not included in the compilation by default, but must be added using the File | Libraries menu option. For full details please see the separate manual LibraryExtensions.pdf included in the Libraries sub-directory of the installation CD.

stdlib functions

char cabs(char v);
char abs(int v);
char labs(long v);
char atoc(char *s);
unsigned char atouc(char *s);
float atof(char *s);
int atoi(char *s);
unsigned int atoui(char *s);
long atol(char *s);
unsigned long atoul(char *s);
int rand();
void srand(int seed);
Header file : <stdlib.h>
Library file : <stdlib.c>

See also Random Numbers.

These functions are provided in the stdlib.c library. This library is not included in the compilation by default, but must be added using the File | Libraries menu option. For full details please see the separate manual [LibraryExtensions.pdf](#).

String Functions

String Functions

Header file : <Strings.h>
Library file : <Strings.c>

See also String Print Functions

These functions operate on strings, they are listed below, please note strings may be in ROM or RAM, but may only be copied to pointers to space in RAM, failure to observe this will result in unpredictable results or a crash. Note that although pointers to unsigned chars are expected, in practice char or unsigned char strings may be used. The functions which start with a lower case r are the same as the normal functions, but the first parameter is always a ram pointer

Function	Return	Notes
<pre> unsigned char Checksum(unsigned char *s); </pre>	Unsigned char	Totals all the characters in the supplied string and returns the value.
<pre> void * memcpy(void *s,void *d,unsigned char c); </pre>	s	Copy c bytes from memory location d to memory location s
<pre> unsigned char * strcat(unsigned char *d,unsigned char *s); </pre>	d	Add string s to the end of string d and return string d
<pre> unsigned char * strchr(unsigned char *d,unsigned char c); </pre>	Result	Find character c in string d and return a pointer to it if found. Return 0 if not found
<pre> char strcmp(unsigned char *d,unsigned char *s); </pre>	-1, 0 or +1	Compares strings d and s and returns 0 if equal, -1 if d is less than s, and +1 if d is greater than s
<pre> unsigned char * strcpy(unsigned char *d,unsigned char *s); </pre>	d	Copy string s to string d
<pre> unsigned char * strncpy(unsigned char *d, unsigned char *s, unsigned char n); </pre>	d	Copy at most n characters from string s to string d
<pre> void vstrcpy(unsigned char *d,unsigned char *s); </pre>	Void	Copy string s to string d, no return
<pre> unsigned char strlen(unsigned char *d); </pre>	Result	Return length of string d
<pre> unsigned char *strlwr(unsigned char *d); </pre>	d	Convert characters in string d to lower case
<pre> unsigned char *strupr(unsigned char *d); </pre>	d	Convert characters in string d to upper case
<pre> unsigned char rChecksum(unsigned char *s); </pre>	unsigned char	Totals all the characters in the supplied string in ram and returns the value.
<pre> void ram * rmemcpy(void ram *s,void *d,unsigned char c); </pre>	s	Copy c bytes from memory location d to

		memory location s in ram
<code>unsigned ram char * rstrcat(unsigned ram char *d,unsigned char *s);</code>	d	Add string s to the end of string d in ram and return string d
<code>unsigned ram char * rstrchr(unsigned ram char *d,unsigned char c);</code>	Result	Find character c in string d in ram and return a pointer to it if found. Return 0 if not found
<code>char rstrcmp(unsigned ram char *d,unsigned char *s);</code>	-1, 0 or +1	Compares strings d in ram and s and returns 0 if equal, -1 if d is less than s, and +1 if d is greater than s
<code>Unsigned ram char * rstrcpy(unsigned ram char *d,unsigned char *s);</code>	d	Copy string s to string d in ram
<code>void rvstrcpy(unsigned ram char *d,unsigned char *s);</code>	Void	Copy string s to string d in ram, no return
<code>Unsigned char rstrlen(unsigned ram char *d);</code>	Result	Return length of string d in ram
<code>Unsigned ram char *rstrlwr(unsigned ram char *d);</code>	d	Convert characters in string d in ram to lower case
<code>Unsigned ram char *rstrupr(unsigned ram char *d);</code>	d	Convert characters in string d in ram to upper case

String Print Functions

```
char *cPrtString(char *String,char n);
char *fPrtString(char *String,float n);
char *iPrtString(char *String,int n);
cha *lPrtString(char *String,long n);
```

```
char ram *rcPrtString(char ram *String,char n);
char ram *riPrtString(char ram *String,int n);
char ram *rlPrtString(char ram *String,long n);
```

Header file : <Strings.h>

Library file : <Strings.c>

fPrtString is described in [Maths Routines](#).

These routines are used to print a decimal representation of a signed number to a string. There are 3 forms, one for characters, one for integers, and one for long types. Use the correct style for the largest type which needs printing as the amount of program space required is smaller for the smaller types.

The functions which start with a lower case r are the same as the normal functions, but the string parameter is always a ram pointer

The number supplied is n. The string must be large enough to hold the complete number when printed. The number is prefixed by a – sign if the number (n) is negative. The string is terminated with a 0 character.

Returns:

A pointer to the string.

Examples:

```
unsigned char Secs;
char s[16];

cPrtString(s,Secs);
```

Wait

void Wait(int Delay);

Header file : <Delays.h>

Library file : <Delays.c>

This function enters a loop for a supplied number of milli-seconds. Delay is the number of milliseconds. The processor undertakes no processing during the delay except for interrupts. Any interrupt occurring during the delay will extend it by the period of the interrupt.

This function is not suitable for timing as delays calling the function are not accounted for. It is mainly of use when the processor is to be delayed for a minimum period of time whilst some other function occurs such as a write to EEPROM.

Returns:

Void function, no return.

Examples:

```
Wait(5); Delay 5mS
```

12 C Reference

Language

Types

Language

Omissions and changes

Extensions

The FED PIC C compiler C language is designed to ANSI standards, the reference source which readers may find most useful is "The C Programming language" by Brian K. Kernighan, and Dennis M. Ritchie, Second Edition, ISBN 0-13-110362-8. This edition of the famous C reference is written to ANSI standards.

The FED PIC C compiler C language has the following omissions and extensions:

Omissions and changes

Functions

The ... form for continuation of function parameters is supported, but only in assembler functions.

Floating Point

The standard version does not include floating point support, this is included in the professional edition. The float and double types create an error in the standard edition.

Include search path

The search path for includes is in the following order :

```
MainDir\LibsUser
MainDir\Libs
MainDir\Libs\ProcIncs
```

Where MainDir is the main application directory. The LibsUser directory is empty and provided to allow users to put in place their own headers, or even to override the FED provided headers.

Library functions

Few of the ANSI library functions are included as not many have relevance to a micro-controller environment. The full list of supported library functions is shown in the [Library Reference](#) section.

Comparisons

If the strict ANSI option is turned off then the result of comparisons will be cast to a signed character (value 0 or 1). If turned on then the result is a signed integer. When turned off compiled files are smaller.

enums

The type of an enumeration is the smallest signed type which can fit the range of results. For example an enumeration of the form:

```
enum Codes {OPEN=0,CLOSED,AJAR};
```

results in a type called Codes which is equivalent to a signed character. However

```
enum BigCodes {OPEN=1000,CLOSED,AJAR};
```

results in a type called BigCodes which is equivalent to a signed integer.

Extensions**Defines****#__config****#asm****#asmdefine****#asmline****#asmend****#asmfunc****#callfunction****#eeprom****#forcequick****#heap****#locate****#locopt****#noheap****#optdup****#optspace****#optspeed****#optquickcall****#preprocdefine****#procfreq****#projectfile****#stack****#usemacro****pointed**

All C extensions are incorporated in the pre-processor, and so are preceded with a '#' character, or are implemented as standard definitions which can be tested with **#ifdef**. To ensure absolute ANSI compliance all extensions can be preceded with **#pragma** which will ensure that other compilers ignore the directive. The extensions are listed below:

Defines*Symbols*

The following symbols or constant integers are defined:

_APPWIZ_AUTO

This value is included and set to the name of the header file which defines the values and elements in a WIZ-C project. To include the header file then simply use # include :

```
#include _APPWIZ_AUTO // Include App Designer header
```

APROCFREQ

This value is included in all compilations by the FED PIC C compiler and is set to the value of the Processor Frequency in the Compile Options Dialog Box. It is also defined in the assembler file (and hence may be tested or used by assembler code) as *_APROCFREQ*.

BANKBITS

This value is included and set to 0, 1 or 2 depending on the number of banks of RAM (1,2, or 4).

_BITTYPES_OPT

This value is defined only when the option to define register bit names without the preceding lower case 'b' is desired. It is defined by the compatibility option "No 'b' in front of bit names". See [Support for 3rd party compilers](#)

COMMON

This value is included and only set to 1 if a common bank of RAM is present (usually from address 70 hex to 7F hex)..

_CORE

This value is included in all compilations by the FED PIC C compiler and is set to 16 for 16 bit core processors (18 series) and to 14 for the 14 bit core processors (12 and 16 series). For compatibility with 3rd party compilers it is better to use *_PIC14* and *_PIC16* (see below).

_FEDPICC

This value is included in all compilations by the FED PIC C compiler and is set to 1.

GPRBITS

This value is included and set to 0, 1 or 2 depending on the number of banks of general purpose RAM (1,2, or 4).

MPC

This value is included in all compilations by the FED PIC C compiler and is set to 1 to indicate that the target is the Microchip PIC family.

__PIC_C__

This value is included in all compilations by the FED PIC C compiler and is set to 1.

_PIC14

This value is set to 1 for a PIC 14 bit core device and is included for compatibility with 3rd party compilers.

_PIC16

This value is set to 1 for a PIC 16 bit core device and is included for compatibility with 3rd party compilers.

__ProcessorName

This value is included in all compilations by the FED PIC C compiler and is set to 1. The name of the define is the name of the processor. E.g. if the 16C84 is selected then the symbol *__16C84* will be defined and set to value 1.

PROCFREQ

This value is included in all compilations by the FED PIC C compiler and is set to the value of the Processor Frequency in the Compile Options Dialog Box divided by 1000. It is also defined in the assembler file (and hence may be tested or used by assembler code) as *_PROCFREQ*.

__PROCTYPE_H

This is defined as a C Macro and is set to the header file name for the current processor. For example if a 16F84 is selected then *__PROCTYPE_H* will be set to the string "P16F84.h" including the inverted commas.

QUICKCALL

This value is included in all compilations by the FED PIC C compiler where the Optimisation Use PIC Call Stack is set and is set to 1. It is also defined in the assembler file (and hence may be tested or used by assembler code) as *_QUICKCALL* and set to 0 or 1 according to whether the option is set to be used.

_ROMSIZE

This value is included in all compilations by the FED PIC C compiler and is set to the size of the ROM area in words for 14 bit core (12/16 series devices), and bytes for 16 bit core (18 series).

_SERIES

This value is included in all compilations by the FED PIC C compiler and is set to 18 for 18 series processors and to 16 for the 12 and 16 series.

_EESIZE

This value is included in all compilations by the FED PIC C compiler and is set to the size of EEPROM data area on the processor. It will be set to 0 for devices with no EEPROM.

false

This value is included in all compilations by the FED PIC C compiler and is set to 0. It is defined as a constant integer.

true

This value is included in all compilations by the FED PIC C compiler and is set to 1. It is defined as a constant integer.

MAXLOCOPTSIZE

This value is included at an assembler level – it cannot be read from C. It is the maximum number of bytes used for local optimisation if all functions are in use.

OVERHEADPAGE0

This value is included at an assembler level – it cannot be read from C. It is the number of bytes used for variables in Page 0 of the device and includes temporary files and bytes used for Local Optimisation. For a 16F, 12F, 12C, or 16C device this will be the number of bytes at address 0x20 used by the compiler (if the device has a duplicate RAM area at 0x70 this is completely used by the compiler and IS NOT included in the count). For a 18C or 18F series device this is the number of bytes used by the compiler for variables at address 0.

Function definitions

Every time that a function is called, a special type of define is created with the same name as the function with a preceding underscore, this can *only* be tested using `#ifdef` in *any* files within the project which *follow* that call. To reiterate, this special define lives beyond the end of the currently compiling file.

For example if the function `SerialIn()` is called and if a subsequent C pre-processor directive of the form:

```
#ifdef _SerialIn
```

is encountered, then it will return true. This is used for library definitions, see section [Creating Libraries](#), for full details of how this is used.

#__config**(or #pragma __config)**

This command defines the configuration register in the hex file, it is supplied with a single word which is the data to be saved for the configuration register, the simulator and some programmers will read this information when they load the hex file. Please note that if the Project | Set configuration options dialog box has been used to set the configuration words then the value set will take precedence over ANY values set

This command works equally well with devices which store configuration fuses at the top of ROM memory, however these devices tend not to have definitions for the config registers.

Example

```
#__config 0x3F3F
```

Where the device has only one configuration register the value is loaded to that register, for example for the 14 bit core, 16Cxx series the address is 0x2007. For the devices in the 18 series with several registers (or 16F devices such as the 16F88) a byte address must be supplied followed by a comma and then the byte value, the example below shows a typical 18 series config set:

```
#__config _CONFIG0,_CP_OFF_0
#__config _CONFIG1,_OSCS_OFF_1 & _HS_OSC_1
#__config _CONFIG2,_BOR_OFF_2 & _BORV_25_2 & _PWRT_OFF_2
#__config _CONFIG3,_WDT_OFF_3 & _WDTPS_128_3
#__config _CONFIG5,_CCP2MX_OFF_5
#__config _CONFIG6,_STVR_ON_6
```

Here is an example for the 16F88

```
#__config 0x2007,0x3F61
#__config 0x2008,0x3FFD
```

Here is an example for a newer 18F series device which stores config values at the top of memory and does not have the values defined in the fuse file. Again note the use of byte addresses and that byte values are provided not word values.

```
#__config 0x300000,0x61
#__config 0x300001,0xFD
```

#asm

(or #pragma asm)

The **#asm** directive starts a block of assembler code. All lines between **#asm** and **#asmend** are taken to be assembler code, or assembler directives, and are written literally to the output file. Please note that the C compiler completely ignores these lines, however any normal pre-processor directives are obeyed, for example code can be included or ignored by using **#ifdef** directives.

Example

The example below shows how to implement a C function in assembler which will clear out the bottom page of RAM on a 14 bit core processor such as the 16C74.

```

void ClearPage0(void);    // Declare function

#asm
#usemacro MRET

Clearpage0::
    movlw 0x20
    movwf FSR
    movlw 0x60
    movwf Temp
cloop:
    clrf 0
    decfsz Temp
    goto cloop
    MRET
#asmend

```

#asmdefine

(or #pragma asmdefine)

The **#asmdefine** directive forces the compiler to define the supplied label at the top of the assembler file before any other assembler code. It is mainly used to force the assembler to include subroutines intended for use by the compiler, but which are being called by assembler. See [Using Assembler](#)

Example

```
#asmdefine _LoadSPD
```

#asmline

(or #pragma asmline)

The **#asmline** directive forces the compiler to pass the rest of the line to the assembler. See [Using Assembler](#)

Example

```

#asmline clrwdt      ; Clear the Watchdog timer
asm("clrwdt");      ; Does the same (compatib)

```

#asmend

(or #pragma asmend)

See [#asm](#)

Also **#endasm** may be used

#asmfunc

(or #pragma asmfunc)

The **#asmfunc** directive is followed by a function name which must have already been defined. It tells the compiler not to attempt to optimise that function.

Example

```
void WriteEEData(unsigned char Addr,unsigned char Data);

#pragma asmfunc WriteEEData
```

#callfunction

(or #pragma callfunction)

The **#callfunction** directive is an internal link to the compiler which ensures that the compiler acts as though the supplied function has been called (this operates to ensure that the correct functions are linked in by the assembler). It is only used in conjunction with assembler routines where the assembler calls a library function. Syntax is :

```
#callfunction Function_Name
```

Example

The example below shows how to direct the compiler to link the code for the function Wait

```
#callfunction Wait
```

#eeprom

(or #pragma eeprom)

This command defines data for the eeprom data area. The **#eeprom** directive is followed by data items. A numeric item is written to the next eeprom data location in turn. If an item is of the form :

```
Address=data
```

Then the data value is written to eeprom address defined.

Example

```
#eeprom 0x3f,8=7
#eeprom 0x9
```

In the example above the values of the first 10 data bytes in eeprom will be :

```
0x3f,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0x07,0x9
```

Note that the eeprom data is embedded in the hex file and will be written to the eeprom data area of devices which have eeprom data by most programmers.

#forcequick

(or #pragma forcequick)

This directive is used in the PIC specific header files to force the compiler to call the specified function using the PIC Call instruction rather than using the software stack – even if quick calling is turned off. It is useful for assembler functions which use return rather than MRET – in particular functions designed to be called from interrupts.

```
#forcequick FunctionName
```

FunctionName is the name of a function which must have been previously declared.

Example

This code defines the serial interrupt handler in the datalib.h file :

```
#forcequick SerIntHandler
```

#heap

(or #pragma heap)

This directive is used in the PIC specific header files to define the value of the heap pointer which is normally defaulted to the first free memory location above variables, or the first byte in the top memory page. **Please note this directive should only be used at the top of the first file in the compilation, failure to observe this may result in a PIC program crash.** Note that this takes preference over the value defined in the Compile Options Dialog Box. See also [Interrupts & Memory](#)

```
#heap Location
```

Location is a constant number which must be either in decimal or 0xnnn hexadecimal format.

Example

This code defines the heap pointer to be at 0x1a0.

```
#heap 0x1a0
```

#locate

(or #pragma locate)

This directive is used in the PIC specific header files to define the internal file registers of the processor as C variables. The syntax is:

```
#locate Variable Location
```

Variable is name of any variable which must have been previously defined, and which must have been defined as type external. Location is a constant number which must be either in decimal or 0xnnn hexadecimal format.

Example

Here is the code which defines the TIMER1 file register for the 16C74 as an unsigned integer located in memory at address 14 decimal, 0E hex, and also as two unsigned characters - one for the lower byte, and one for the upper byte.

```
extern unsigned int TMR1;
extern unsigned char TMR1L;
extern unsigned char TMR1H;
#locate TMR1 0xe
#locate TMR1L 0xe
#locate TMR1H 0xf
```

#locopt

(or #pragma locopt)

This directive is used in the PIC specific header files to set the number of local optimisation bytes used by the compiler. Note that this takes preference over the value defined in the Compile Options Dialog Box. **Please note this directive should only be used at the top of the first file in the compilation, failure to observe this may result in a PIC program crash.** See also [Optimising Your Output](#)

```
#locopt value
```

value is the number of bytes to use for local optimisation. value is a constant number which must be either in decimal or 0xnnn hexadecimal format.

Example

This code instructs the compiler to use 8 bytes for local optimisation.

```
#locopt 8
```

#noheap

(or #pragma noheap)

This directive is used with application programs (such as Real Time Operating Systems) which do not allow for the use of the heap. The directive does nothing other than force an error if the heap is used when the #pragma noheap directive has been encountered. The heap is only used for storing interim results which are too large for the internal accumulators such as when a structure is returned by a function.

Example

```
#noheap
```

#optdup

(or #pragma optdup)

This directive is used to turn on, or off the duplicate block optimiser. The selected state is used for the rest of the file until the next optdup directive. Note that if the duplicate optimiser is turned off for the project (in the options dialog), then it will not be turned back on by "optdup 1". See also [Compiler Options Dialog Box](#)

```
#optspace value
```

value is either 0 or 1 depending on whether the option should be turned on or off.

Example

This code instructs the compiler to optimise for space.

```
#optspace 1
```

#optspace

(or #pragma optspace)

This directive is used in the PIC specific header files to set the compiler to optimise the output for space saving. Note that this takes preference over the value defined in the Compile Options Dialog Box. **Please note this directive should only be used at the top of the first file in the compilation, failure to observe this may result in a PIC program crash.** See also [Optimising Your Output](#)

```
#optspace value
```

value is either 0 or 1 depending on whether the option should be turned on or off.

Example

This code instructs the compiler to optimise for space.

```
#optspace 1
```

#optspeed

(or #pragma optspeed)

This directive is used in the PIC specific header files to set the compiler to optimise the output for speed. Please note this directive should only be used at the top of the first file in the compilation, failure to observe this may result in a PIC program crash. Note that this takes preference over the value defined in the Compile Options Dialog Box. See also [Optimising Your Output](#)

```
#optspeed value
```

value is either 0 or 1 depending on whether the option should be turned on or off.

Example

This code instructs the compiler to turn off optimisation for speed.

```
#optspeed 0
```

#optquickcall

(or #pragma optquickcall)

This directive is used in the PIC specific header files to set the compiler to generate output to use the PIC call stack when calling functions. **Please note this directive should only be used at the top of the first file in the compilation, failure to observe this may result in a PIC program crash.** Note that this takes preference over the value defined in the Compile Options Dialog Box. See also [Optimising Your Output](#)

```
#optquickcall value
```

value is either 0 or 1 depending on whether the option should be turned on or off.

Example

This code instructs the compiler to use the PIC call stack for function calling.

```
#optquickcall 1
```

#preprocdefine

(or #pragma preprocdefine)

This directive is used to define a value which will be included as a C macro definition in all files following this one which are compiled. As library files are always compiled last the directive is a way of communicating to a library in C.

```
#preprocdefine MacroName=Substitution
```

This will have the same effect for all following files as if the following C definition were included at before the compilation of each file :

```
#define MacroName Substitution
```

Example

This code defines the Vendor ID for the USB library :

```
#pragma preprocdefine _VendorID=0x04d8
```

Now within the USB code the following can be written to test and use the earlier definition:

```
#ifndef _VendorID
    const int vid=_VendorID;
#else
    const int vid=0x000;
#endif
```

#projectfile

(or #pragma projectfile)

This directive is used to include files in the project file window and is useful for libraries to ensure that correct files are included in the build. A file with a .txt extension will be included as a comment file, a file with a .sti extension will be included as a simulation file, all other files will be included as C source files.

Any file defined will be added to the project window if it is not already present. The file name should be enclosed in inverted commas (") and unlike any other C string back slashes should not be shown as \\ but should be shown as a single \. If a full path is not given then the current project directory is assumed. Finally \$(FEDPATH) at the start of the string will be replaced by the directory of the main program executable.

For example :

```
#pragma projectfile "$(FEDPATH)\libs\usb\cdc\sti\configure.sti"
```

This example ensures that `configure.sti` is included in the project, assuming the standard install location then this will be expanded to :

`c:\program files\fed\pixie\libs\usb\cdc\sti\configure.sti.`

`#procfreq`

(or `#pragma procfreq`)

This directive is used in the PIC specific header files to define the value of the processor frequency. Please note this directive should only be used at the top of the first file in the compilation, failure to observe this may result in a PIC program crash. Note that this takes preference over the value defined in the Compile Options Dialog Box.

`#procfreq value`

value is a constant number which must be either in decimal or 0xnnn hexadecimal format.

Example

This code defines the processor frequency to be at 2.5MHz.

`#procfreq 2500`

`#stack`

(or `#pragma stack`)

This directive is used in the PIC specific header files to define the value of the stack pointer which is normally defaulted to the top location in memory. **Please note this directive should only be used at the top of the first file in the compilation, failure to observe this may result in a PIC program crash.** Note that this takes preference over the value defined in the Compile Options Dialog Box. See also [Interrupts & Memory](#)

`#stack Location`

Location is a constant number which must be either in decimal or 0xnnn hexadecimal format.

Example

This code defines the stack pointer to be at 0x7F.

`#stack 0x7f`

`#usemacro`

(or `#pragma usemacro`)

Still operates for backward compatibility. Defunct do not use in new projects.

```
#usemacro macro text
```

pointed

Note that for processors with more than 32K words of program the standard 2 byte pointer can only point to the bottom 32K of ROM memory. To allow pointers to address functions in this situation any function which may be called through a pointer should be defined as pointed :

```
void pointed MyCallableFunction(int param) ;
```

Therefore there is a limit imposed on the program that no more than 32K of functions may be called through a pointer. In practice this limit is far greater than required - if more than this size of program is required then small functions can be defined which call bigger functions.

There is no need to use the pointed keyword for processors or programs which utilise less than 32K words of program memory.

pointed is not standard ANSI, to allow programs to be portable the following can be defined at the top of the source file :

```
#ifndef _FEDPICC
#define pointed
#endif
```

Types

Built in types

Types defined in PIC header files

Built in types

bit

The **bit** type is a single bit and consequently has some limitations. It is very efficient for holding flags in global memory as it can be set, reset, and tested with a single instruction and only occupies one bit of memory so that 8 bits only take one byte. Note this is a non-ANSI type. bit behaves like an unsigned char of length 1 bit so it takes the values 0 or 1.

The limitations of the bit type are that it cannot be used as a function parameter – the following example is illegal:

```
void func(bit x) ;
```

It's address cannot be taken and therefore arrays of bits cannot be created either. As a local variable in a function it is automatically defined as unsigned char, and therefore its use as a local variable is also not recommended as its behaviour may not be as expected.

char

The **char** type is the fundamental type supported by the PIC, 8 bits. Signed values can represent results from -128 to +127, Unsigned from 0 to 255.

HINT

Use the unsigned char type wherever possible as it is the most efficient form for both program space and execution speed. The unsigned char type is also defined as a type in all the PIC headers which is called BYTE.

int

The **int** type is 16 bit type which is slower and uses more code space than the **char** type. Signed values can represent results from -32768 to +32767, Unsigned from 0 to 65535.

long

The **long** type is a 32 bit type which is slower and uses more code space than the **char** or **int** types. Signed values can represent results from -2,147,483,648 to +2,147,483,647, Unsigned from 0 to +4,294,967,295.

float and double

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.

The **double** type is the same as float at preset.

Pointers

Pointers declared as ram (e.g. **char ram *xp;**) are one byte in length and may only point to items in the bottom 256 bytes of address space for 12 and 16 series devices. For 18 series devices these pointers are 2 bytes long, but use the FSR accumulator and are considerably more efficient, they may only point to items in RAM space.

All other pointers are 2 bytes in length allowing any location in File Register or Program memory to be addressed. To differentiate between File Register (RAM) and program memory the top bit is set. Thus the pointer 0x0006 points in File Register memory to address 6 which is PORT B on most processors, the address 0x8006 points to address 6 within program memory. Please note that if any pointer to Program Memory is de-referenced then the program will expect that location to contain a RETLW instruction, any other instruction may cause a program crash.

Note that for processors with more than 32K words of program the 2 byte pointer can only point to the bottom 32K of ROM memory. To allow pointers to address functions in this situation any function which may be called through a pointer should be defined as pointed :

```
void pointed MyCallableFunction(int param);
```

Therefore there is a limit imposed on the program that no more than 32K of functions may be called through a pointer. In practice this limit is far greater than required - if more than this size of program is required then small functions can be defined which call bigger functions.

HINT

For 12 or 16 series processors define all pointers to items in the bottom 256 bytes of memory to be of type ram.

For 18 series devices define pointers which will only ever be to RAM to be of type ram.

Types defined in PIC header files***BYTE***

This type is defined in all the PIC headers and is equivalent to **unsigned char**.

FileReg

This type is defined in all the PIC headers and is equivalent to **unsigned char**.

ulong

This type is defined in all the PIC headers and is equivalent to **unsigned long**.

WORD

This type is defined in all the PIC headers and is equivalent to **unsigned int**.

13 Pre Processor

Many thanks to Marcel Van Lieshout for his excellent work during 2004 on the pre-processor.

The pre-processor is encapsulated in a DLL, it is called WIZCPP, this section of the manual describes the pre-processor. The pre-processor runs before the main compiler and produces an intermediate file which includes only compiler specific options (using #pragma).

Specifications

WIZCPP pragma Options

Specifications

WIZCPP and the ANSI99 standard

Pragma

WIZ-CPP options

Linelength

Assembler and the pre-processor

sizeof() operator

#include

Magic defines

WIZCPP and the ANSI99 standard

WIZCPP is aimed to be fully compliant to the ANSI99 standards specification. There are some deviations from this standard implemented. These deviations were necessary to optimize the preprocessor to the wiz-C integrated development environment.

Pragma

Compliant with the standard, pragma's are copied onto the output. Their output always starts in column 1 of a new line. Only the pragma will be on this line. Before writing the output, a macro-expansion is performed. It is therefore perfectly legal to write a pragma like:

```
#define _PROCFREQ 20000
#pragma procfreq _PROCFREQ
```

A special pragma is defined to set options for WIZCPP itself. The pragma-identifier is WIZCPP-compile-time defined. The default value is "WIZCPP". WIZCPP-pragma's will never be written onto the output. An example of a WIZCPP-pragma:

```
#pragma WIZCPP writecomments off
```

For further explanation of the WIZCPP-pragma's: see "WIZCPP options".

WIZ-CPP options

Next to pragma's WIZCPP also accepts other #-directives. An example:

```
#pragma asmfunc myfunc

can also be written as:

#asmfunc myfunc
```

Of course WIZCPP has to handle these, too. This is an addition to the ANSI99 standard. The options are converted by WIZCPP into real pragma's and handled as such (see above). The compiler will therefore only see real pragma's which will always start in column 1.

Linelength

WIZCPP can be configured (WIZCPP-compile-time) to limit the output linelength. This is introduced to safeguard the compiler from running out of bufferspace. Normally an error is raised when the outputlinelength is exceeded and the remainder of the line will not be written. It is possible to configure WIZCPP (WIZCPP-compile-time) in such a way that all macro-expansions will write a newline character when encountering a backslash/newline combination. This will greatly reduce the risk of exceeding the maximum linelength, but is definitely not ANSI99-compliant as it will break lines where it normally should not. The default-setting for linelength is 1024. Linebreaks are disabled by default to maintain ANSI99 compliance.

Assembler and the pre-processor

Using pragma's or wiz-C options, it is possible to enter assembler directly into C-source. WIZCPP is aware of these pragma's as macro-expansion needs to be suspended, each assemblerline needs to start on a new line and comments are structured differently from C-comments. This awareness of compiler pragma's by the preprocessor is an addition to the ANSI99 standard.

Because ANSI99 states that pragma's (or any other #directive) are not allowed within macro-definitions, "#pragma asm" cannot be used to include assembler in a macro definition. To embed pragma's into macro-definitons, use the ANSI99 _Pragma() operator instead. An example:

This is correct:

```
#define nop() \
    _Pragma("asm") \
    nop \
    _Pragma("asmend")
```

or:

```
#define nop() _Pragma("asmline nop")
```

But this won't work:

```
#define nop() \
    #asm \
    nop \
```



```
#asmend
```

sizeof() operator

WIZCPP recognizes the sizeof() operator for basic types. This is not the sizeof() operator that is handled by the compiler. The sizeof() operator can be used in expressions on #if and #elif lines to conditionally compile code dependant on basic typesizes:

```
#if    sizeof(int) == 2
    #define INT_MAX          32767
#elif sizeof(int) == 4
    #define INT_MAX          2147483647
#else
    #error "Unable to determine integersize"
#endif
```

The following basic types are supported: char, short, integer, long, float and double. Pointers to these types are supported as well as pointers to functions.

#include

The ANSI definition of the #include directive is fully supported. One addition has been made: When using the #include directive with a token as argument (as opposed to "" or <>), the token does not need to have delimiters. A WIZCPP-compile-time option is available to activate or de-activate this addition. Another WIZCPP-compile-time option defines the delimiter that is to be used. The wiz-C extension is enabled by default, the default delimiter is "".

```
// These are ANSI compliant:

#include      "myheader.h"
#include      <sysheader.h>
#define headerfile <anotherheader.h>
#include headerfile

// The wiz-C addition also allows:

#define headerfile  anotherheader.h
#include headerfile
```

Magic defines

The following magic defines (also known as dynamic defines) are supported:

__LINE__	The linenumber, currently being processed (eg. 146)
__FILE__	Full name of the current sourcefile (eg. "C:\dir\main.c")
__FILENAME__	Filename of the current sourcefile (eg. "main.c")
__PATHNAME__	Pathname of the current sourcefile (eg. "C:\dir\")
__CURRPATH__	The setting of the currentpath pragma
__DATE__	The current date (eg. "Feb 22 2004")
__TIME__	The current time (eg. "11:04:19")

`__TIMESTAMP__` Full date/time (eg. "Sun Feb 22 11:04:19 2004")

`__STDC__` "Are we standard C"-flag (always 1 for wizC)

`__STDC_VERSION__` ANSI standard to which we comply (always 199901L for wizC)

The dynamic defines can eg. be used to create more descriptive error messages:

```
if(error) {
    fprintf(stderr, "Weirdness on line %d in file %s\n",
        __LINE__, __FILE__);
    exit(1);
}
```

or include the date and time of compilation into the object:

```
const char *timestamp = __TIMESTAMP__;
```

WIZCPP pragma Options

All WIZCPP-options are implemented as `#pragma`'s. They can appear anywhere in the inputstream. They only affect WIZCPP's behaviour from the inclusion point on. By using `pragma`'s, ANSI99 conformance is maintained while still being able to finetune WIZCPP to a specific task without the use of (sometimes rather cryptic) commandlineparameters.

The available options are described in alphabetical order.

Currentpath

Debug

Exitnormal

Expandasm

Expandnl

Includeoptional

Keepfiles

Linelength

Magicdefines

Psizes

Searchpath

Sizes

Trigraphs

Uselib

Writecomments

Writesyncs

Currentpath

```
#pragma wizcpp currentpath path
```

With this option, a directory can be set to be searched for `#include`-files (mostly headerfiles .h). It only affects `#include`-directives where the filename is enclosed within `""`. Only one directory can be set: A new `currentpath`-directive will overwrite the current path. The *path* must be enclosed within `""` and should end with a `\` (backslash).

Debug

```
#pragma wizcpp debug debuglevel
```

This option activates or deactivates the generation of debuginformation to the outputfile. The outputfile will therefore no longer be a correct C-file as all kinds of additional information are written into it. As the option is only used during WIZCPP debugging, this should not cause a problem. The *debuglevel* can be set to a value of 0 thru 9, where 0 disables debugging (default) and 9 is the highest level. Every level includes the output of all lower levels. At every use of this directive the internal symboltable is dumped. When WIZCPP is compiled without debug-support, this #pragma is ignored.

Exitnormal

```
#pragma wizcpp exitnormal
```

When WIZCPP encounters any warning, error or fatal (= cannot continue) condition, it will exit with an exitcode of nonzero to indicate to it's parent process that errors were found. When the *state* of this option is set to 'on', WIZCPP will always exit with a zero (= no errors found) exitcode. Possible values for *state* are 'on' and 'off'. The default state is 'off'.

Expandasm

```
#pragma wizcpp expandasm state
```

Normally, wizcpp expands assemblerlines in the same way as normal C-code, When the *state* of this option is set to 'off', wizcpp suspends the expansion of assembler. Possible values for *state* are 'on' and 'off'. The default state is 'on'.

Expandnl

```
#pragma wizcpp expandnl state
```

The ANSI specifications define that a combination of a backslash and a newline is to be treated as line-concatenation: It glues two lines (or more if several consecutive lines are each terminated by a backslash-newline pair) together into one long line. By default, wizcpp adheres to this specification.

It is possible, however, to configure wizcpp in such a way that all backslash-newline pairs will write a newline character when encountered. This will greatly reduce the risk of exceeding the maximum linelength of the wizC compiler, but is definitely not ANSI99-compliant as it will break lines where it normally should not.

The most common use of this pragma is when large macro's need to be handled. The standard way of expansion can lead to very long lines being generated by the preprocessor. These long lines may overrun the maximum allowed linelength for the compiler. By using the expandnl pragma, one can selectively choose to wrap these long lines into multiple shorter lines.

To be able to nest multiple occurrences of this pragma, an internal stack of 100 levels keeps track of the history of this pragma. To enable expansion while retaining the previous setting(s), use:

```
#pragma wizcpp expandnl save,on    // save current setting, then
enable
// linebreaks
```

Restore the setting to the state before this pragma invocation, use:

```
#pragma wizcpp expandnl restore    // restore previous setting
```

Includeoptional

```
#pragma wizcpp includeoptional filetoinclude
```

This directive allows the optional inclusion of a (eg header-) file. Normal ‘#include “”’ or ‘#include <>’ syntax and processing is used except that no warning/error is raised when the file cannot be found. A possible use is the inclusion of additional settings (eg. searchpaths) on a per-project basis.

Keepfiles

```
#pragma wizcpp keepfile state
```

Setting *state* to ‘on’ prevents WIZCPP from deleting the inputfile and errorfile after a succesfull (= no fatals, errors and/or warnings) run. . The default state is ‘off’. When WIZCPP is compiled with debug-support, the files are always kept.

Linelength

```
#pragma wizcpp linelength length
```

Wizcpp can be configured to limit the output linelength. This is introduced to safeguard the compiler from running out of linebufferspace. An error is raised when the outputlinelength is exceeded and the remainder of the line will not be written. Setting *length* to 0 (zero) disables the check. The default *length* is 250 characters.

Magicdefines

```
#pragma wizcpp magicdefines state
```

The *magicdefines* option allows turning ‘on’ and ‘off’ the recognition of the dynamic defines (eg. __LINE__ and __FILE__). Possible values for *state* are ‘on’ and ‘off’. The default state is ‘on’.

Psizes

```
#pragma wizcpp psizes list
```

With this option the sizes are defined that are used by the sizeof()-operator when it evaluates the size of a pointer. *List* is a comma-separated enumeration of the sizes (in bytes) of the various basic pointertypes. The order of the sizes is fixed and cannot be changed. The *list* must have exactly seven elements. The order is: (char *), (short *), (integer *), (long *),

(float *), (double *) and (*)(). The last entry defines the size of a pointer to a function. The default sets all pointersizes to 2 bytes (2,2,2,2,2,2,2).

Searchpath

```
#pragma WIZCPP searchpath path
```

With this option, a directory can be added to the list of directories to be searched for #include-files (mostly headerfiles .h). One directory can be added per searchpath-directive. The maximum number of directories allowed is defined during WIZCPP-compilation. The *path* must be enclosed within delimiters, either "" or <>. Although the chosen delimiter does not affect the searchorder, it might be helpful for documentary purposes. The order all directories are searched is the order that the directives appear in the inputstream. When a relative path is used, the current path (see #pragma WIZCPP currentpath) is prepended.

Note that the searchpath is cleared at the start of each C file.

Sizes

```
#pragma wizcpp sizes list
```

The sizeof()-operator allows a programmer to write conditional code based on the size of the basic C-types. Because of that, the preprocessor has to be aware of the sizes of these basic types. *list* is an enumeration of the sizes in specific order. Because six basic types are supported and recognized, *list* must contain exactly six entries (seperated by comma). The order of definition is : char, short, integer, long, float and double. The default sets the *list* to 1, 2, 2, 4, 4, 4.

Trigraphs

```
#pragma wizcpp trigraphs state
```

ANSI99 defines a way to use frequently used characters in C-source when these characters do not appear on the keyboard. This method is called "trigraphing". Currently there are nine trigraphs defined in the standard. The nine trigraphs and their replacements are:

Trigraph:	??(??)	??<	??>	??=	??/	??'	??!	??-
Replacement:	[]	{	}	#	\	^		~

Possible values for *state* are 'on' and 'off'. The default state is 'off'.

Uselib

```
#pragma wizcpp uselib library-to-include
```

Through the use of this option, an application can instruct WIZ-C to add a library to it's list of libraries. The library-to-include must be enclosed within "". When a relative path is used, the current path (see #pragma wizcpp currentpath) is prepended. It is possible to use macro's

within *library-to-include*. To differentiate between path-elements and macro's, the macro's should be preceded by a \$. This pragma is especially usefull for writing easy-to-use libraries: The pragma will then be present in the headerfile accompanying the library.

Writecomments

```
#pragma wizcpp writecomments state
```

Normally all comments are stripped from the output of the preprocessor. Sometimes, mostly during debugging, it comes in handy to pass all comments on to the output. Setting *state* to 'on' will do just this. The default setting is 'off'.

Writesyncs

```
#pragma wizcpp writesyncs state
```

To allow a simulator or ICD-device (in-circuit-debugging) to synchronize to the original sourcelines, WIZCPP is capable of writing synchronization-lines to it's output. These lines contain the sourcelinenummer and the filename the line originally came from. The writing of these lines can be enabled or disabled by using this option. Possible values for *state* are 'on' and 'off'. The default state is 'on'. The structure of the synchronization-lines is defined during WIZCPP-compilation.

14 Use with MPLAB

FED recognise that MPLAB is the tool of choice for debugging PIC applications for many users, although we believe that the in-built debugger is faster, easier to use and more flexible for software debugging, MPLAB has In-Circuit Emulation and Debugging facilities not yet available in PICDE.

MPLAB

Integrating with MPLAB poses a number of problems when considering the FED PIC C/PICDE model. MPLAB does not support long filenames. MPLAB only supports a single file for Assembler projects and only integrates command line tools, not complete development environments such as those from FED. MPLAB does not support DDE or any other forms of automation. Finally MPLAB will only single step in the assembler file - the original source is shown as comments in the assembler file.

FED have overcome most of these problems. PIC C has a menu option under the tools menu called "Run MPLAB (C)". When the project is compiled and assembled PIC C also produces a special ASM file for MPLAB which has all the macros stripped out and replaced by the code within the macros - this file has the same name as the project name, but with `_MPL.ASM` appended. The code in this file is identical to the code assembled by PIC C, but is in a single file. PIC C also produces a project for MPLAB called with the same name, but with a `.PJT` extension.

When the Run MPLAB (C) option is clicked PIC C runs MPLAB with this project. If it is already open the project will need to be re-opened.

Procedure for using MPLAB

For reasons connected with the conversion of long filenames to short file names, FED recommend that projects intended for intensive use with MPLAB should have a file name which is 4 characters or less in length, and should not be buried too deeply in sub-directories. If MPLAB reports that a file cannot be opened then resave the project in a higher level directory. As usual FED recommend that each project should have its own directory.

First use of MPLAB with a project

Use the PIC C compiler and compile and assemble your project. Click the **Tools | Run MLAB (C)** menu option. MPLAB will be opened with this project. When the project is opened in MPLAB it will create a warning that the hex file cannot be found. Ignore the warning and press F10 to assemble the project. Now reload the project from the MPLAB project menu (by using the Project menu, and then pressing key 1). *Do not save the project* - this will allow the watch variables to be loaded properly.

Subsequent use of MPLAB with a project

Use the PIC C compiler. Click the **Tools | Run MLAB (C)** menu option. PICDE will start, assemble the files, and then open MPLAB. If MPLAB is already running with the project open then PICDE will request the user to close the project, and then re-open the project manually.

When the project is opened in MPLAB press F10 to assemble it.

Debugging

MPLAB will have only one file as assembler source - this will contain the complete source code without any macros or conditional assembly, but with the original C Code as comments. When the project is opened from within FED PIC C, a breakpoint is automatically generated at the label "main", - the first statement of the C program. This means that the Debug Run (F9) option can be used to run to the first statement used. Following this it is recommended that the right mouse key is used on the main source file to bring up the MPLAB menu - the option "Run To Here" can be used to skip through C statements.

PIC C will automatically include a watch window called "PICDE" within the MPLAB environment, this will contain all the debug variables defined in PIC C. This window is re-defined every time that the **Run MPLAB (C)** menu option is used so it is recommended that a different Watch window is used for user defined variables.

PIC C also transfers all of its unconditional addressed breakpoints into MPLAB.

MPLAB hex file

Please note that the hex file produced by MPLAB will have the filename XXX_MPL.HEX where XXX is the project name. If XXX is more than 4 characters then MPASM will produce a short file name which will be in the form XXX_M~1.HEX.

For this reason FED recommend that project files for intensive use with MPLAB should be of 4 character length or less.

MPLAB crashes ?

Occasionally MPLAB appears to crash when transferring from PIC C. In this case create the project manually - create a new project from MPLAB with a new file name. Include one Assembler node for this project - the file will have the same name as the PIC C project with an _MPL added. For example if the PIC C file is called Test then the assembler file which MPLAB can open is called Test_MPL.asm.

15 The Professional Version

Introduction to the professional version

Familiarity

Installation

MultiProject Management

Assembler Projects

Viewing and inspecting variables

History

Waveform Generator

Introduction to the professional version

Welcome to the professional version of WIZ-C and the FED PIC C Compiler.

The enhancements which are available in the professional version allow the user to:

- Manage and simulate multiple projects together
- Connect PIC pins across projects to allow simulated devices to communicate
- Handle assembler and C projects
- View variables in native C format
- View a list of all local variables and their values
- Maintain a history within simulation to back track and determine the past leading up to an event

Familiarity

It is expected that users of the professional versions of our Integrated Development Environments are already familiar with the use of the program and have run through the introductory tutorial.

Installation

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.

PLEASE NOTE – if upgrading an existing copy then all project files stored within the directory structure should be backed up – either to an external storage device, or elsewhere on the hard disk.

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 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.

MultiProject Management

The most powerful upgrade within the professional version is the ability to manage and simulate two or more projects together using PIC's which may be communicating. These projects may use the application designer or switch it off, and can be C or assembler based in a free mix.

It is quite possible to simulate the same code running in two or more communicating devices, the example shows a good use of this capability.

Within the professional version it is possible to open and manage a project in the same way as for the normal version. When a single project is opened it behaves in the same way as the normal version. Note that projects are referred to as "Project Groups" in the professional version – this is to allow for additional projects to be added and managed.

If a an additional project is opened (using the **Project | Add Project to Group** menu option) then instead of closing the current project, then a new or existing project is added to the project window. The project window will now contain two or more tabs – one for each project. Clicking on a project makes that project the *Current Project*. Each tab contains a list of the files used for that project.

The project group is saved together under the project file name of the first project in the group. For example if the first project is called "BusMaster" and there is one additional project called "Bus Slave" then if the BusMaster project is opened the group will be opened and both BusMaster and BusSlave projects will appear in the project window. If the BusSlave project is opened then it will be opened on its own.

Each project has its own debugging window although the editor and information windows are shared between all the projects. The tool bar also contains a list of projects, each with its own button. Clicking a project tab in the project window, or clicking the button in the toolbar will make that project active, it will also bring the debugging window for the project to the top of the stack of windows. This is a quick way to look at the activity of a particular project.

Actions which normally act on a project will now either act on every project, or on the current project.

Project Options. Project options for the active project are set using the **Project | Current Project Options** menu command. Options are set only for the one, current project.

Compiling using Control and F9 will now compile all the projects, one after another. If any project contains an error then the compilation of all projects will stop at the project with errors.

Simulation. By default all projects are simulated together, when the run command is used all projects will run together at the correct speed, this means that some projects may execute more instructions than others in the same time if they have a higher clock rate.

It is possible to simulate only the Current Project during development. This is a switch on the Simulate menu option – **Simulate All Projects Together**. By default this is checked. When it is cleared then only the current project is simulated which is faster.

When a breakpoint is hit in any projects then all projects cease to run in the simulator.

When the processor is reset then a breakpoint is set at the main function of all C projects and each is run together. The first project to hit the main function will stop all projects from running, therefore some projects may halt in the initialisation code. If the simulation is run again then each project will halt the simulation in turn as they hit the main function. This can be avoided by using the **Simulate | Absolute Reset** menu option which starts all projects at address zero.

The single step and step over commands apply to the current project. All other projects will also run and all will halt when the single step or step over command is complete on the current project. With differing processor speeds this implies that when one project is single stepped other projects may run no instructions, or may run two or more instructions in the same time.

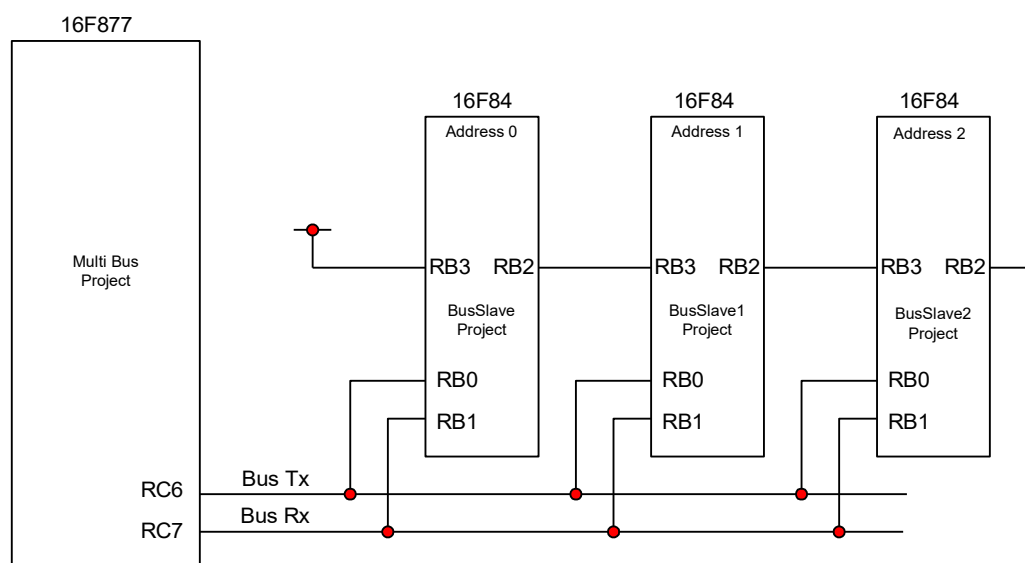
Example Project #1 – Multiple project management and simulation.

For the professional version we will not look at development of a multi-project group from scratch, but instead will look at an example project group and how it is managed. It is assumed that users will be familiar with the use of the FED integrated environment and will be able to create the multi project group using the familiar principles of the normal version.

In the example project we will look at the development and simulation of a complete program using WIZ-C FED C users may still open and examine the program in the same way, but will not be able to use the Application Designer front end. This is not a hindrance to using and understanding the project.

The program we will look at is designed for a 16F877 processor as a bus master with three 16F84 devices as bus slaves. It is assumed that the 16F84 devices are performing processor or I/O intensive tasks which communicate back to the master 16F877 device on request.

The circuit block diagram is shown below:



The 16F877 is operating as a bus master, it use asynchronous communication to send a receive bytes from the three devices on the bus. Each of the 16F84 devices is running the same program.

At power on the f877 waits for 100mS to allow all devices to power up. It then sends a 0 byte on RC6.

The F84's need to determine their addresses, this is done by a simple algorithm. At power on they set the RB2 pin to output at level 0. They wait until the 0 byte is received from the bus master. This serves to synchronise the devices. Once the 0 byte is received they enter an enumeration routine. If the RB3 input is high then the device knows it is address 0. It wait 1mS and then raises RB2 high. If the RB3 input is low the device adds one to its address, waits 10mS and then samples it again. It repeats this until the RB3 is high at the end of a 10mS period when it knows that its address is set, it then waits 1mS and raises its RB2 output. As each device sets its address it raises RB2 and the next device knows its address at the end of the next 10mS period.

During normal operation the F84's wait for commands from the F877. The F877 sends two bytes, the first is a device address, the second is a command byte. If the command byte is 'A' then the addressed device simply sends back a 'K' character to verify its presence. If the command byte is 'B' then the addressed byte sends back a message byte.

For the test program the F877 uses Timer 1. When Timer 1 overflows (about once every 250ms), then the F877 polls each of up to 16 devices to request them to return their message byte. The Message bytes are stored in an array.

For the test program each F844 sets its message byte to its address as a test value. The message byte is incremented every time that the F877 requests the message byte.

When simulating we will be looking for each F84 to set its address on reset and then the Message Array in the F877 to hold the message received from each F84 which will be incremented on each bus read.

The project is developed with WIZ-C using the asynchronous and timer elements.

Opening the project group

Use the **Project | Open/New Project Group** command. Look in the Projects folder within the FED compiler folder. Within this folder is a sub folder called MultiProject. Within this folder are four projects. The project group is stored under the master project called Multi-bus. Open this project.

Note that this project group contains four projects, MultiBus which is the F877 project, and three projects BusSlave, BusSlave1 and BusSlave2. These projects correspond to the 16F84 devices shown in the block diagram above. Each of these projects is in fact based on the same core code

There are two ways of handling multiple projects based on the same core code. Firstly develop a project which is needed for simulation in two or more devices. The best method to create a new device is to create a new project with a new name using **Add Project to Group**. Turn off the application designer for this project by clicking the project tab to make it the current project, and then use the **Project | Use Application Designer** menu option to

turn off the application designer. Now simply add all the files from the first project to the second in the same order and the two projects will compile the same code. The method we have used here is slightly different to illustrate the use of the application designer in more than one project. We have created 3 projects each with their own Application Designer, set up in the same way with the same elements and events. We have then included into each project the same file “BusSlaveCommon.c” which contains the occurrence handling code.

Click on each button on the toolbar to bring each project to the front – look at the application designer, the project window and the debugging window for each.

Setting options and compiling, resetting

Click on a toolbar button of one of the projects. Use the Project | Current Project Options menu option to see how to set the project options for just one project.

Use Ctrl+F9 (or **Compile | Generate Application**) All four projects will be compiled one after the other. All being well there are no errors.

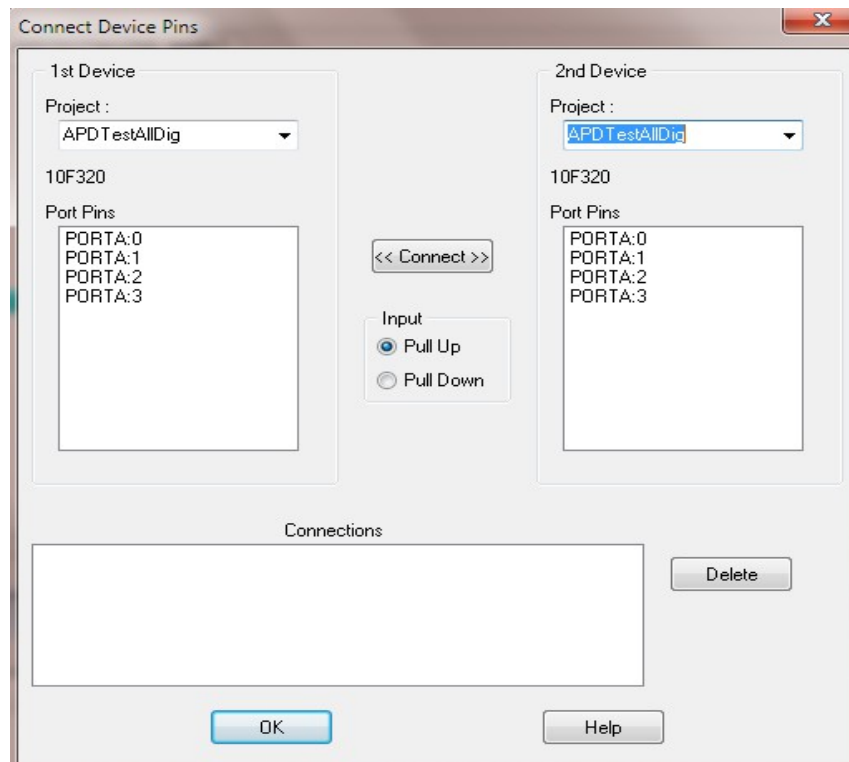
After compiling the simulator will (as usual) reset all of the PIC's. However it will run to the main function on the 3 slave processors before it hits main on the F877. Click the project buttons along the top to see the execution point (a blue bar) in each processor. Note that all of the bus slaves are at the same address – main(), however the F877 (MultiBus) is still initialising.

Run the program and this time the program will stop as the F877 hits main at which point the slave processors will all be in the main loop.

Linking Device Pins

Being able to simulate projects together would be of little use if it were not possible to link PIC pins. The simulator allows PIC pins to be linked. The simulator behaves as if PIC pins are linked with a low value resistor so that if both PIC's drive at the same time the value on the pins of the PIC is the level driven by that PIC. It is also possible to assume High value Pull Up or Pull Down resistors on each link so that when the pins are not driven the link takes a logic level.

The pins are linked using the **Simulate | Link PIC Pins** menu option. The Connect PIC Pins dialog box is created.



To connect two pins the first project is selected in the left hand Project box and the second project in the right hand box. Select a pin from the first device and a pin on the second, choose pull up or pull down resistors and click the Connect button.

The link will be shown in the connections box at the bottom. A link may be deleted by selecting it and clicking the Delete button. Note that every connection is shown bi-directionally and both ends of the link need to be deleted. It is possible just to delete one direction from a two way link in which case the simulation behaves as though a diode were in the link.

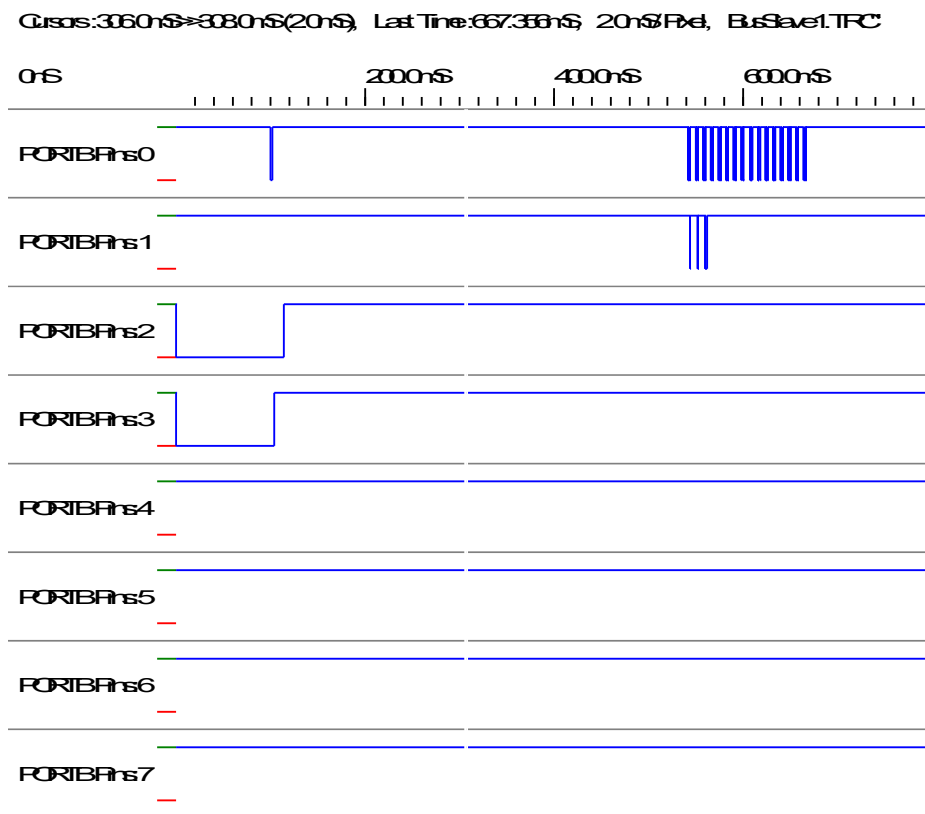
Experiment with creating new links, but do not delete any of the existing links at present. Note that the F877 is connected to the same pins on each of the F84's and each F84 is connected to its neighbour – follow the links from the project block diagram shown above.

Simulating the main project

Run the simulation until about 1 second. Note that the simulator will be slower than normal as it is running four devices in parallel, and handling inter-device communication.

Refer to the block diagram above.

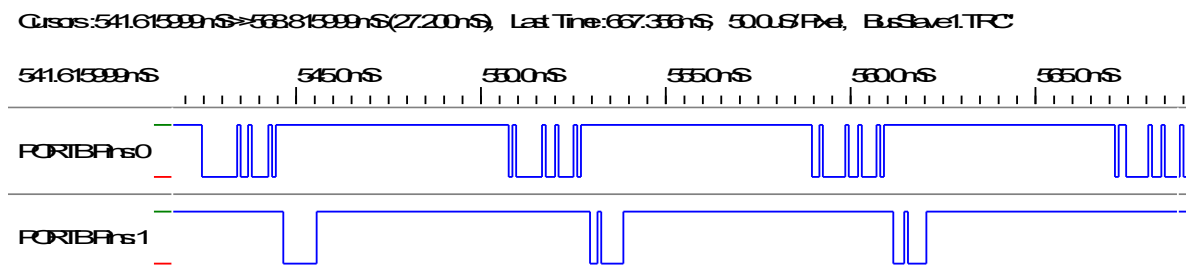
Now click BusSlave and then the other two slave projects. Note how the address variable has been correctly set. Note also the message variable which will be different for each project – it is set to the address multiplied by two initially and incremented each time that the value is read by the F877. Click BusSlave1 and examine the wave analyser for that project. Examine PORTB as 8 line traces. You should see the following display:



Recall that PORTB:0 is an input to the 16F84, and an output from the F877, and PORTB:1 is the input to the F877 and an output from each F84.

The byte received at about 100mS from the F877 is the initial 0 byte to set off the address determination function. We are looking at BusSlave1 so there will be a short delay of 1mS before PORTb bit 3 (the input from BusSlave 0) goes high, 10mS later this device detects the high signal, sets its address, and pushes its own output on bit 2 high to BusSlave2. This is the delay seen between Port B:2 and B:3 going high.

Now at about 500mS the F877 starts the first of its regular polls of all the attached slaves to receive the message from each. 16 devices are polled. The first 4 polls are shown below in an expansion of the above waveform:



Note that the F877 sends two bytes at a time (using asynchronous protocols), the address which is seen here as the bytes 0, 1, 2, and 3. Following this is the byte 'B' (hex 0x42) which is sent to each device and which commands each device to return its message. The response

seen from the first 3 devices is the message which is the address multiplied by 2. (0, 2 and 4 in this case). No further devices are attached to the bus so the simulation shows no further responses.

Now click the MultiBus button at the top of the screen. Look at the debugging window. It shows the list of messages received in an array of 16 values. Dependant on when the simulation was stopped, it should show 0, 2 and 4 as shown above followed by a number of 0xFF values which is the value written when a device is not detected. Run the program and watch as the F877 reads the messages which increment on every reading.

You may like to look at the source files to see how the program is working – as for most WIZ-C programs the actual amount of source code required is quite low.

Assembler Projects

The professional version of WIZ – C now allows any project in the group, or any individual project to be an assembler project which may then be assembled and debugged in the same environment.

Assembler projects should follow these guidelines:

The application designer should be turned off.

The file type in the Project File box (selected when the files are added to the project) should be C/H/ASM.

The compiler detects an assembler project by the absence of any files with a C extension, so provided that no files have a C extension then the project will be assembled rather than compiled and no C library code will be included.

The ICD cannot be used with WIZ-C assembler projects

As an example of a very simple assembler file which will operate correctly in WIZ-C, here is a program to constantly increment PORTB:

```
#include <p16f84.inc>

    org 0

    clrf STATUS
    clrf PORTB
    bsf STATUS,RP0
    clrf TRISB
    bcf STATUS,RP0

CLoop    incf PORTB
        goto CLoop
```

ASM files in C projects

It is also possible to include ASM files within a C compilation. These files act as though the compiler directives #asm and #asmend have been used to bracket the code, and the files are assembled in order with the C files in the order shown in the project window.

Viewing and inspecting variables

The professional version includes a considerably improved variable inspector. The compiler now saves information for the simulator to enable it to determine the scope and type of variables available within the program at all points within the source file.

When a variable is added to the watch window it is now possible to select it as being a C source variable. The “Use C Definition” check box in the top left of the Debug Watch dialog box will select a C variable. (Recall that adding a variable to the debug watch window is most easily accomplished by pressing insert when the window is active, or using the Add Watch option of the menu which appears when the window is right clicked).

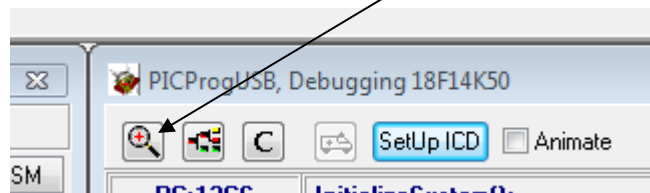
C Variables are shown with a small C symbol besides them:

C x M 0x64

For integer values the display format may be chosen as hex or decimal. The simulator will attempt to show the entire contents of the variable and dereference pointers. If the program moves to a new function or block where the variable name has a new meaning, then the C variable will always be shown as the value which is in scope.

Viewing Local Variables

The entire list of local variables active at any time during the program may be shown by showing the Local Variable list. This is turned on or off by right clicking the watch window and using the “**Inspect Local Variables**” menu option. Or by using the main **Simulate | Inspect Local Variables** menu option. Or by using the button on the debug window:



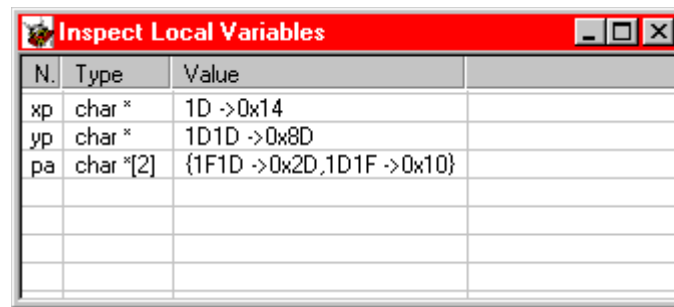
Please note that users no longer need to be aware of where local variables are held (stack or optimised memory), or how they are offset from the stack – the simulator determines all this automatically.

The example below shows local variables for this function:

```
void ptrtest()
{
    char ram *xp;
    char ram *yp;

    char ram *pa[2];

    //
    // More code here
    //
}
```



N.	Type	Value
xp	char *	1D ->0x14
yp	char *	1D1D ->0x8D
pa	char *[2]	{1F1D ->0x2D, 1D1F ->0x10}

Note how the array and the pointers have all been de-referenced. Pointers may also be de-referenced correctly even if they point to ROM based constant values.

Inspector Windows

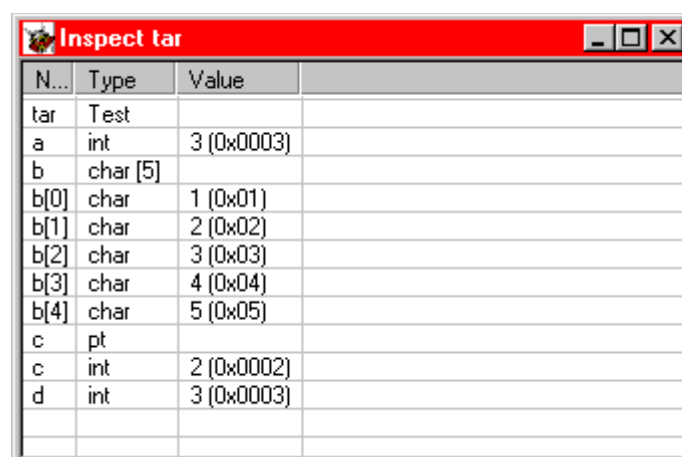
Inspector windows allow the entire contents of a variable to be shown in detail. Consider the following code:

```
struct pt
{
    int c,d;
};

struct Test
{
    int a;
    char b[5];
    pt c;
} tar;
```

Now to inspect the value of the tar variable in detail on a pop up window then the **Simulate | Open Inspector** menu option may be used (alternatively position the cursor over a variable and use Shift F2, or use the right pop menu on the debug window when a watch variable is selected).

This is the result for the this example:



N...	Type	Value
tar	Test	
a	int	3 (0x0003)
b	char [5]	
b[0]	char	1 (0x01)
b[1]	char	2 (0x02)
b[2]	char	3 (0x03)
b[3]	char	4 (0x04)
b[4]	char	5 (0x05)
c	pt	
c	int	2 (0x0002)
d	int	3 (0x0003)

Note how the variables within the structure have been shown in type and detail.

Inspector windows float over the main window and so are always available for examining values whatever the main program is doing.

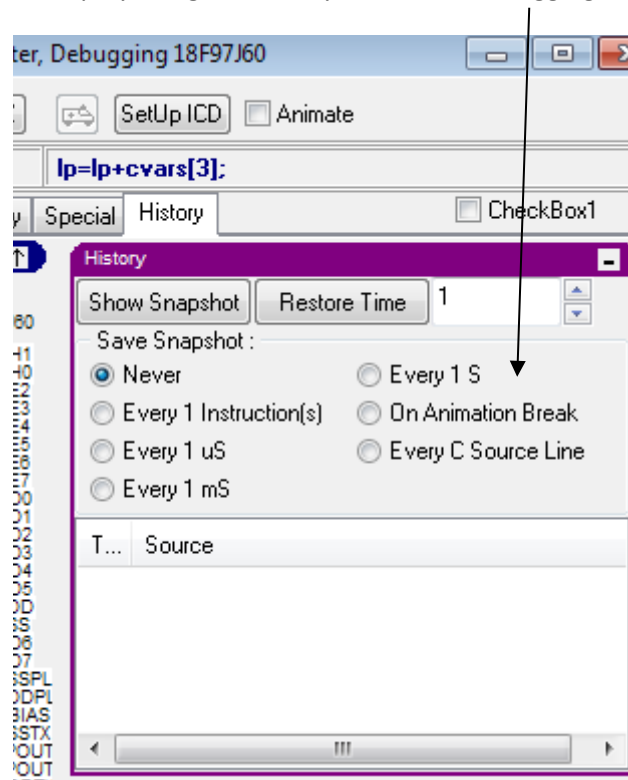
Inspector windows are not saved with the project and so must be opened as required during simulation.

History

The history facility keeps a record of the state of the PIC over the past leading up to the current point. It is very useful for tracking errors, or checking what led to a crash, or reviewing the state of the PIC at specific points – particularly if a very long simulation is to be running when post simulation review can be much faster than waiting for the next breakpoint.

Setting history parameters

The history facility is set up by using the history tab on the debugging window :



The Save snapshot list gives the opportunity to select when to take a snapshot of the state of the PIC for later review. The options are as follows:

- **Never** Default. No snapshots will be taken as the program simulates. Note that snapshots do slow the simulation.
- **Every n Instruction** A snapshot is taken every time that the nth instruction is executed. The number n is set using the box to the right of the snapshot list.
- **Every n uS** A snapshot is taken once every n microseconds. The number n is set using the box to the right of the snapshot list.
- **Every n mS** A snapshot is taken once every n milli-seconds. The number n is set using the box to the right of the snapshot list.
- **Every n S** A snapshot is taken once every n seconds. The number n is set using the box to the right of the snapshot list.

- On Animation Break If a breakpoint is hit which has the Animation box clicked then the debugging window is updated. At the same time a snapshot may be taken. This is very useful for keeping a record of the state of the PIC at specific points in the program such as on completion of a function which is of interest.
- Every C Source Line A snapshot is taken every time a C source line is executed

Examining history

Once the program has stopped, either manually, at a breakpoint, or when an error occurs then it is possible to look back in the simulation. The History tab contains a list of snapshot times together with the source line at which the snapshot was taken. Double click a time to take the simulation back to that time. The edit window will show the position of the program counter at that time in the source file, the watch tab will show the values of memory at that time, and any open inspector windows – including local variables – will show the value of those variables at that time.

To move back and forward through history points use ALT plus F5 to backwards to the previous snapshot time and ALT plus F6 to go forward. ALT F4 returns to the current time. The simulation will also return to the current time if the program is stepped or run. Note that the time box in the left of the toolbar will show the time of the snapshot – it will also turn red when a snapshot is being examined, and returns blue when showing the current normal time.

Controlling the number of snapshots

There is a limit to the total number of snapshots as each takes space in memory. The normal limit is 100 – the file options dialog box (Menu **File | Options**) allows this to be changed..

Example of use of snapshots

To look at using snapshots we'll use an example of a recursive function which causes an error during simulation.

The function we'll use is a simple recursive factorial:

```
float Factorial(float n)
{
    float x;
    if (n==1.0) return n;
    x=Factorial(n-1);
    return n*x;
}
```

This is certainly not the best or most efficient way of calculating factorials, but it does illustrate the use of History points.

Open the project History\Factorial.PC in the Projects folder of the main program. The project uses a 16F877 and simply determines the factorial of 5 followed by the factorial of 50. Click the History tab of the debugging window. Check that it is set to save a snapshot on every C source line. There should also be a breakpoint after the program has determined the factorial of 5 (if not click the mouse in the left margin of the main function by the line "Result=Factorial(50);" to set a breakpoint.

Run the program – it should calculate factorial 5 and stop. Check the watch tab to see that the value of Result is 120. Click the history tab and you should see that the program has executed about 20 lines of C code - note how the program runs the line “return n*x” four times in succession at the end. Double click a line to see the program state at that point. Click the watch tab and use the Alt F5 and Alt F6 keys to move back and forward in time – note how the value of the local variables change in the Local Variables window and also see the values on the watch window.

Now run the program again. Factorial(50) would take 300 bytes of software stack space if it were to be successful which is far too big for the F877 and so the program will fail.

This time it will stop with an error resulting from stack underflow of the PIC stack. Look at the history window again. If you click the oldest value in the window – which should be around 1mS, then watch the value of sp (the software stack pointer) on the watch tab as you move forward in time using Alt F6 . sp will reduce as the recursive function is called and as it does so watch the value of n. When sp reduces below 80 Hex then the software stack has overflowed (it is now pointing to system variables) – n starts taking on corrupt values and the program is now doomed to a final crash which happens a surprisingly long time later.

This example shows how it is possible to use history to look back in time to see the point where a program fails and what causes a simulation error which as in this example may be some considerable time before the simulation fails.

Waveform Generator

The Simulator Wizard is intended to allow users to design complex data and analogue patterns for injection to the pins of the device under simulation.

Introduction

The waveform wizard is a front end for the FED PIC and AVR development tools. It is available with the professional version of these tools.

The wizard allows complex data patterns to be input to the PIC, clocks to be generated, or analogue waveforms to be generated for injection into the A/D converter inputs of the PIC. The waveform wizard allows a number of stimulus' to be stored together in one file. One of more of these files may be added to the list of project files and will then be included as simulation input when the program is simulated.

Starting a project

Within any of the FED development environments use the **Tools | Run Wave Generator** menu option to start the program. A new wave generator project will be created with the same name as the current project. The wave generator file will be added to the project window which will automatically include the stimulus when the program is simulated.

Double click the wave generator file in the project window to start the wave generator and load that file.

Alternatively start the Wave Generator using the start menu. A new project can be created using the **File | New** menu option. Save the project using the **File | Save As** menu. Now the project may be added to IDE project window by right clicking the project window and using the "Add | Insert Item menu option. Select the wave generator project file name and ensure that the Wave Generator type is selected

Waveform Generator help

The waveform generator has its own help file and manual which should be read separately.

Example

There is an example of the use of the Wave generator provided in the projects folder.

Open the project "WaveGenTest" in the folder of the same name in the Projects folder. This project performs a number of simple functions. It uses a serial element connected to PORTB bit 0 and bit 1 (using interrupts to receive bytes on that bit), it has an A/D converter input on pin A1 (A/D converter channel 1). It converts the value on A1 to an 8 bit value which is written as an 8 bit digital value to PORTD. Any byte read from the serial port is simply retransmitted. Finally it uses a 10KHz clock on the Timer 1 input which is used together with the capture/compare CCP1 to count the passing of seconds by resetting Timer 1 when it reaches 10000.

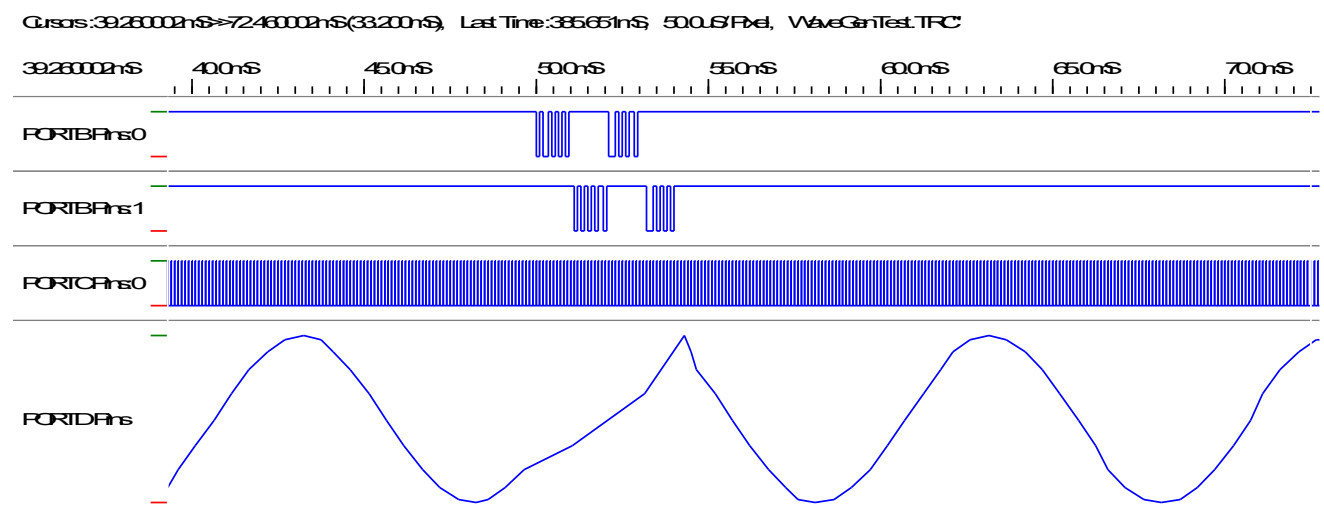
Double click the WaveGenTest.STM file in the project window to open the Waveform Generator. Note that there are 3 stimulus inputs, Analogue, Clocks and sti_0. Select each using the Edit Stimulus drop down box and explore the parameters of each.

The *Clocks* stimulus injects a 10KHz clock into pin RC0 which is the Timer 1 count input.

The *Analogue* stimulus generates a 100Hz Sin Wave injected into Port A bit 1.

The *sti_0* stimulus generates two bytes – 0x55 and 0xaa which are injected into Port B bit 0. Note these are separated by a byte period.

Return to the WIZ C window and run the project for a couple of seconds, note how the variable SecCnt increments once a second. Open the waveform window (**Tools | Open Wave Window**). Explore the waveform – here are the patterns seen on various ports of the PIC:



Note that there is a fairly smooth Sin wave on PORTD which is the digital value converted from input RA0. However as the serial port uses a software algorithm the PORTD output is corrupted as serial bytes are received on RB0 and retransmitted on RB1. This could be avoided by using the interrupt driven serial port.

Note the 10KHz clock input to RC0 – this is set to have a Mark/Space ratio of 25%.

16 Command Line interface

It is possible to call WIZ-Professional from the command line from where a number of switches may be used to control its operation including completely automatic operation. The syntax is :

```
pixie.exe /c /e /m /s /w /x filename.pc
```

e.g. `pixie.exe /c /e /m /s /x testprogram.pc`

filename.pc should be a full path.

This example opens PIXIE to compile the project called testprogram. It opens PIXIE with a minimal window. If compilation is successful the compiler will exit immediately. If it fails the compiler will remain open so that the error can be investigated.

The switches are all optional, they *must* be separated by spaces or tabs and are described in detail below:

/c	This option autocompiles – as soon as the project has loaded it will be compiled.
/e	This option causes the compilation to stop if an error is found and leave the compiler running so that the error may be investigated. It overrides the /x option when errors occur.
/m	This option causes the compiler to run in a minimal window – the program loads and displays the information window only. This is used to show progress.
/s	Silent mode – dialog boxes automatically close. This mode may be used to start the compiler, auto compile and exit without any user intervention. If used with the /e or /w option then errors will cause a dialog box to be displayed.
/w	This option displays a warning if errors are found during compilation.
/x	This option forces the compiler to exit after completing an auto compilation.

The return value from the program is the number of errors found, or 0 if none are found. Successful compilation may be tested by checking the return value, or by checking if the .hex file in the Output directory exists.

17 List of library functions

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	<u>LCD</u>
LCD	<u>LCD</u>
LCDString	<u>LCDString</u>
LN2	Maths Routines
LN10	Maths Routines
log	Maths Routines
log10	Maths Routines
LOG2_10	Maths Routines
lPrintToString	String Print Functions
memcpy	String Functions
pClockDataIn	<u>ClockDataIn</u>
pClockDataOut	<u>ClockDataOut</u>
PI	Maths Routines
pow	Maths Routines
pow10	Maths Routines
PowerSeries	Maths Routines
printf	printf Functions
pSerialIn	<u>SerialIn</u>
pSerialOut	<u>SerialOut</u>
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
strchr	String Functions
strcmp	String Functions
strcpy	String Functions
strlen	String Functions
strlwr	String Functions
strupr	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	<u>Wait</u>
WaitRx	Interrupt Driven Serial Port
WriteEEData	EEPROM Routines

18 List of 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