48623 Mechatronics 2

University of Technology Sydney

Notes on Embedded C

Ver 1.3 by Dr Sarath Kodagoda, Dr Damith Herath and Mr Michael Behrens

INTRODUCTION

This short note set provides the basics needed to complete the Mechatronics assignments in Mechatronics 2 subject. This by any means is neither a comprehensive note nor would it replace reading a good reference book on the subject. Some suggested references are listed at the end for the more enthusiastic reader.

Microcontrollers have generally been programmed using machine language. However due to the inherent complexity, higher level languages such as Basic, Pascal and C have become common place in programming such devices. This short primer will provide some basic guidelines for programming microcontrollers using C.

CONTENTS

1. C Variables

1.1 Introduction

Variables store user data in sections of the memory. Prior to using a variable it should be declared (ie. Given a name) along with the matching type (ie. Type of data expected to be stored inside the variable).

Variable names must start with a letter or underscore, then letters, numbers and the underscore '_' can be used. Spaces are NOT allowed. Certain keywords that are part of the C language are reserved and should not be used as user defined variables. These keywords are: auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while.

Variables are case sensitive (ie temp, Temp are not the same). Only the first 31 characters are significant.

Syntax:

```
type   variable_name;
```

Eg:

```
char counter_1;
```

Variables should be initialised before they are used. If a variable is not initialised it will have a random value when first used which can cause the program to malfunction. Initialisation is achieved by writing a known value to the variable. Often initialisation is performed when the variable is defined.

Eg:

```
char counter_1 = 0;
```

1.2PIC VARIABLE TYPES

The commonly used PIC variable types are as follows,

Type	Description	Example usage
bit	A bit can store a Boolean varable (ie. 0 or 1).	bit error_flag;

int	An int is 16 bits wide and stores signed numbers between -32,768 and +32,767.	int temperature;
		temperature = -15;

unsigned int	An unsigned int is 16 bits wide and could hold decimal numbers in the range 0 to +65,535.	Unsigned int count;
		count = 0;

unsigned char	8 bits wide and can store a single ASCII character or an integer number in the range 0 to 255.	unsigned char tag;
		tag = 'D'

long	A long is 32 bits wide and holds large signed numbers in the range -2,147,483,648 to 2,147,483,647.	long addition;
		addition = 23554;

unsigned long	An unsigned long type is 32 bits wide and holds large unsigned integer numbers in the range 0 to 4,294,967,295.	unsigned long cycle;
		cycle = 295294967;

float	A float data type is 24 bits wide in the range 1.2E-38 to 3.4E38 and is used to store no integer fractional numbers (i.e. floating-point real numbers) implemented using the IEEE 754 format.	float volt;
		volt = 4.95;

double	A double data type is 24 bits or 32 bits wide (selected during the compilation) and is used to store double-precision floating-point numbers.	double dist;
		dist = 21.12345

1.3 STATIC VARIABLES

Static variables are usually used in functions and can only be accessed from the function in which it was declared. The value of a static variable is retained on exit from the function and becomes available again when the function is next called.

Static variables are declared by preceding them with the keyword static,

Eg:

```
static int count;
```

1.4 Volatile Variables

A variable should be declared volatile if its value can be changed by outside events beyond the control of the program in which it appears. An interrupt service routine is one such example.

Variables shared by the main code and interrupt service routines as well as variables used for I/O should be declared volatile.

Eg.

```
volatile int volt;
```

1.5 Persistent Variables

Normally, when a program starts up for the first time or after a reset is applied to the microcontroller the initial values of variables are cleared to zero. This could be prevented by using the persistent qualifier.

For example, the variable limit is declared persistent and as a result its value is not cleared after a reset:

Eg:

```
persistent int limit;
```

1.6 Absolute Address Variables

Absolute address variable holds the address of an absolute location in the microcontroller memory. The character '@' after the name of the variable is used to define the address. For example,

Eg:

```
int Led @ 0×05
```

where the variable Led is assigned the absolute hexadecimal address 0x05.

2. C Functions

2. 1 Introduction

Functions help to achieve smaller tasks within a complex program. They enable reuse of code and modularization. Appropriate functions hide details of operation from parts of the program that don't need to know about them, making it easier to understand the overall program and simplifying the process of making changes within.

Built-in functions are the general purpose functions that are built in to the complier. User functions are created by the programmer to address the specific needs of the project involved.

2.2 Built-in Functions

Built in functions are native to the compiler used and are generally available in compiler related documentation.

Eg:

cos, sin, sqrt

2.3 User Functions

As the name suggest, user functions are developed by the programmer to address particular needs that are not covered by the compilers in-built functions. Every function has a name and optional arguments, and a pair of brackets must be used after the function name in order to declare the arguments. A function can return values or can be declared void, so no values are returned after the execution of the function as in the following example,

```
void motor_off()

{
    speed = 0;
}
```

Following example illustrates how multiple values could be passed to function and the results returned to the calling function. Here two integers, x and y are passed to the function add, the two values are summed inside the function and the result (z) is then returned to the calling function

```
int add(int x, int y)
{
    int z;
    z = x + y;
    return(z);
}
```

This function could be called in the program to do various additions. For instance to add two numbers 10 and 12 we call the newly defined function,

```
int a;
a = add(10, 12);
```

Also note that the variables defined inside a function are local to that function and do not have any relationship to any variables used outside the function with the same names. In the previous example, the variable z defined inside the add function is not accessible from the outside and the programmer is free to define z in the main program for a different use.

If a programmer wants a particular variable to be accessible to all parts of the program the variable should be declared as a global variable. The syntax for declaring a global variable is the same as the syntax for declaring a local variable however the declaration is located outside the main function and before all other functions. The programmer must ensure that global and local variables are not declared using the same name.

3. Program Flow control

3.1 Introduction

For any complex software program, flow control is necessary. It allows for certain parts of the program to be executed repeatedly as in a loop, or certain parts to be executed only when certain conditions are met or to jump from one section of the program to another. The main structures used in flow control are discussed below.

3.2 If-Else Statement

This statement is used when a particular section of the program should only be executed when a certain condition/s are met.

The syntax is as follows for a single statement;

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

and for multiple statements,

```
if(condition)
{
    statement;
    . . .
    statement;
}
```

The if statement could be followed by the else statement to execute an alternative set of statements when the if(condition) is not satisfied.

Eg.

```
if(condition)
{
    statement;
    . . .
    statement;
}
else
{
    statement;
    . . .
    statement;
}
```

Some implementation examples are given below,

Note 1: see how the curly brackets are used when multiple statements need to be executed (above) and they are avoided when a single statement is executed.

Note 2: The text followed by // are called comments and are ignored by the compiler. They are used to help the programmer understand the code. This is discussed in detail in section 5.3.

```
if (volt > 3.0) flag=1;     // if the value of volt is greater than 3 flag is set to 1


if (volt > 3.0) flag=1;
else flag=0;     //if value of volt is less than or equal to 3 flag set to 0
```

3.3 For Statement

In order to create a loop inside the program, the for statement could be used. The general for loop format is as follows,

```
for (initial_value; condition; increment)
{
    statement;
    . . .
    statement;
}
```

Loop starts with the initial_value and continues until the condition becomes false. The third parameter is executed at the end of each loop;

An example where a counter is implemented using a for loop to count from 0 to 9 is shown below;

```
for (i=0;  i<10; i++)  count=i;          // variable count is assigned the value of i each
                                         // time the loop executed.ie 0 to 9.
```

3.4 While Statement

Similar to the for loop discussed previously, the while statement could also be used for creating loops in a program. While loop is especially useful when the number of loops to be executed is not known before entering the loop. The format of the while loop is shown below;

```
while(condition)
{
    statement;
    ...
    statement;
}
```

The loop repeats until the condition at the beginning of the statement becomes false.

Some example implementations are shown below;

The following example is similar to the counting example in the previous section;

```
i = 0;
while(i < 10)
{
    count = i; //note the assignment comes before the increment of i.
    i++;
}
```

See if you can figure out why the following loop never ends.

```
i = 0;
while(i < 10)
{
    count =i;
}
```

Hint: the variable i is always less than 10.

The while loop is regularly used in microcontroller programming in order continuously monitor interrupts etc. A simple infinite loop could be created for this purpose as follows;

```
while (1)
{
    RB0 = 1;
    DelayMs(1000);
    RB0 = 0;
    DelayMs(1000);
}
```

3.5 Do Statement

This is another form of the while statement where the condition to terminate the loop is tested at the end of the loop and, as a result, the loop is executed at least once. The general format of this statement is:

```
do
{
    statement;
    ...
    statement;
} while(condition);
```

The 0…9 counter example could be implemented using the do loop as follows

```
i = 0;
do
{
    count = i;
    i++;
} while(i < 10);
```

3.6 Break Statement

The break statement could be used to terminate the execution of a block of code. The 0…9 counter is implemented using the break statement in the following example;

```
i=0;
while(1)
{
    count = i;
    i++;
    if(i==10) break;  //Note: for the equality check '==' is used.
}
```

3.7 Switch-Case Statement

Similar to the if-else statement, but has more flexibility to accommodate multiple branches depending on the condition of a testing variable. The switch–case statement can only be used in certain cases where:

- only one variable is tested and all branches depend on the value of that variable;
- each possible value of the variable can control a single branch.

Implementation of this could be illustrated using a simple example. Assume there is a variable count which could take values 1,2 & 3 and according to the value a different LED (ie. RB0, RB1 and RB2) is lit. This could be implemented using switch-case statement as follows;

```
switch(count)
{
    case 1: //count=1
        RB0=1; //turn on the first LED
        RB1=0;
        RB2=0;
        break; //this terminates the block and jumps out of the switch-case block
    case 2:
        RB0=0;
        RB1=1;
        RB2=0;
        break;
    case 3:
        RB0=0;
        RB1=0;
        RB2=1;
        break;
    default: // executed if count does not take one of the expected values
        RB0=0; //turn off all LEDs
        RB1=0;
        RB2=0;
        break;
}
```

4. PRE-PROCESSOR COMMANDS

4.1 INTRODUCTION

Pre-processor commands extend the capabilities of the language. These commands are usually added at the beginning and are identified by the hash '#' symbol appended to the front.

4.2 #INCLUDE

This is the most commonly used pre-processor command. This command causes the pre-processor to replace the line with a copy of the contents of the named file. For example,

```
#include <myproject.h>
```

or

```
#include 'myproject.h'
```

causes the contents of file myproject.h to be included at the line where the command is issued.
When developing programs with the PICC Lite compiler, the following line must be included at the beginning of all programs:

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

The file pic.h includes all the PIC microcontroller register definitions.

4.3 #DEFINE

This command is used to replace numbers and expressions with symbols. Some examples are given below:

```
#define PI 3.14159

#define MAX 10

#define MIN 0
```

Note that there are spaces between the words and no ; at the end of the lines.

When these symbols are used in the program their values are substituted to where the symbols are. For example,

```
area = PI * r * r;
```

is changed by the pre-processor into

```
area = 3.14159 * r * r;
```

4.4 #IFNDEFINE

This command in tandem with above #define and #endif provides conditional preprocessing.

For example, to make sure that the contents of a file myproject.h are included only once, the contents of the file are surrounded with a conditional statement like this,

```
#ifndef MYPROJECT_H

#define MYPROJECT_H

statement;

...

statement;

#endif
```

4.5 #ASM

The #asm command identifies the beginning of assembly code followed by #endasm at the end of the assembly code. For example,

```
#asm
mov   ah,09
#endasm
```

The programmer can also specify single assembly instructions using the following format.

```
asm(“instruction”);
```

This format is most often used to include a no operation (NOP) instruction. For example,

```
asm(“nop”);
```

5. OTHER C ESSENTIALS

5.1 ARRAYS

An array holds multiple variables of the same data type. For example, instead of having,

```
int volt1 =12;
int volt2 = 5;
int volt3 = 2;
```

an array could be defined to hold all these values such that,

```
int volt[3];//define the structure
volt[0]=12; //assign values to the variables
volt[1]=5;
volt[2]=2;
```

Each indexed value of the array are accessed using square brackets (ie. x[i]) and the indexing starts with 0, so that x[2] refers to the third element in an array called x.

Arrays can also store characters. In the following example, the characters MECHATRONICS could be stored in a character array such as:

```
unsigned char course_name[12] = {'M', 'E', 'C', 'H', 'A', 'T', 'R','O','N','I','C','S'};
```

5.2 STRUCTURES

A Structure could be thought as a collection of records, a method to collect related items together. For example, the details of a student could consist of,

```
unsigned char name[60];
unsigned int student_number;
unsigned char address[100];
```

This could be realized through a structure as follows:

```
struct student // define the structure called 'student'
{
    unsigned char name[60];
    unsigned int student_number;
    unsigned char address[100];
};
```

Once a structure is declared it could be used to store data by declaring variables of that type (ie in above example student) as:

```
struct student first_student_data; //this will hold data of the first student
```

The elements of a structure can be accessed by writing the structure name, followed by a dot, and the element name. For example, the student_number of structure first_student_data can be assigned number 100123456 as:

```
first_student_data. student_number = 100123456;
```

5.3 Comments

Comments in a program are statements that are not executed during runtime but useful in providing clarification and improving readability of code. Comments are indicated by either, // for single line comments or /* … */ pair for multiple line comments. We have already seen the first form of comments used in previous examples.

Example

```
//this is a single line comment
i = 10; //initialize i

/* add i and x in order to calculate
the total. This is a multiple
line comment.
*/
total = i + x;
```

5.4 ARITHMETIC AND LOGIC OPERATORS

Following operators are commonly used in C.

()[]	parenthesis
!	logical NOT
~	bit complement
+-*/	arithmetic add, subtract, multiply, divide
%	arithmetic modulo (remainder from integer division)
++	increment by 1
--	decrement by 1
&	address of
<<	shift left
>>	shift right
>	greater than
<	less than
sizeof	size of a variable (number of bytes)
==	logical equals
!=	logical not equals
||	logical OR
&&	logical AND
+=	add and assign
-=	subtract and assign
/=	divide and assign
|=	logical OR and assign

^=	logical NOT and assign
&=	logical AND and assign
>>=	shift right and assign
<<=	shift left and assign

5.5 Number Bases

Following are the valid number bases,

Number base	**Format**	**Example**
Binary	0bnumber	0b10101011
Octal	0number	0253
Decimal	number	171
Hexadecimal	0xnumber	0xAB

5.6 INTERRUPTS

Interrupts are part of any real-time program. Most PIC microcontrollers offer internal and external interrupt facilities. Internal interrupts are usually timer-generated interrupts, while the external interrupts are triggered by activities on the I/O pins.
When an interrupt occurs, the program jumps to a special function where the interrupt is handled called an interrupt service routine (ISR). ISR must have the name interrupt followed by a user-selected function name, and the function must not return any value and must not have any arguments.
For example, an ISR function named isr that toggles the state of a variable called RB0 could be written as:

```
void interrupt isr(void)
{
    RB0 = !RB0;
}
```

Interrupts must be enabled before they can be recognized by the microcontroller. The ei() and di() instructions enable and disable global interrupts, respectively. Additionally, the interrupt control bit of each interrupt source must be enabled in the appropriate special function register (e.g. TMR2IE bit must be set to 1 to enable Timer2 interrupts).

It is important to keep the execution time of the ISR as short as possible so that the disruption to the main execution of the program is minimised. This is usually achieved by setting flags within the ISR which are checked as part of the main loop. If the particular flag is set when the main program checks it the appropriate is taken. This allows the code to be executed in an orderly and timely manner.

It is also important to refrain from jumping out of the ISR function using instructions such as goto as this would prevent the interrupt from finishing and cause the program to fail.

6. STRUCTURE OF A C PROGRAM

6. 1 INTRODUCTION

Having covered the main components of the language, now we can put them together to write practical code.

6. 2 MAIN

Any C program needs to include the main{} function, which acts as the starting point of program execution. A simple program to add two integers could thus be written as,

```
main()
{
    int total;
    total = 1+3;
}
```

6. 3 A PRACTICAL EXAMPLE

However, for a program to perform truly useful and real world functions, code needs to include supporting functions, external source files, comments and other compiler includes. Following is an example of a program:

```
/ ***********************************************************
Program Add
```

```
Written by: A. Student
Date: 1 – 1 – 2009
File: add.c
This program adds two integer numbers
*******************************************************/

#include <pic.h> // this file provides the necessary interfacing functions

//program word
__CONFIG(HS & WDTDIS & PWRTEN & BOREN & LVPDIS & DUNPROT & WRTEN & UNPROTECT);

// supporting functions
int add_two_numbers(int x, int y)
{
    int z;
    z = x +y;
    return(z);    //return the resulting value back to the calling function
}

// Main Function
main()
{
    int a, b,c;   //declare variables with type integer
    a = 10;       //assign values
    b= 45;
    c = add_two_numbers (a,b); // call the supporting function
}
```

7. Interfacing PIC

7.1 Introduction

In this section we will try to understand the basics of interfacing a PIC microcontroller with external devices and how to write simple code to 'talk' to such devices.
Since your projects are based on the PIC 16F877A chip, examples are given for this particular chip. We will use the MPLAB IDE and HI-TECH Picc-Lite compiler to write, debug and compile code.

7.2 PIC Interface

Following schematic shows the pinout diagram for the PIC controller used in your projects,

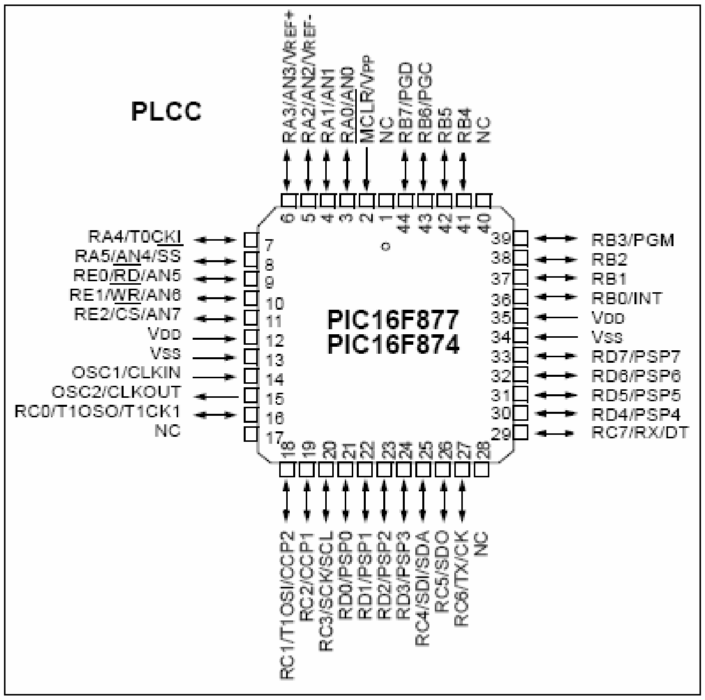


Fig 1 PIC 16F877 Pin-out diagram

In order to interface with external devices, the input-output ports (ie RA, RB, RC, RD, RE) of the PIC controller should be connected with these devices.

7.3 A Simple LED Flasher

Best way to understand how to program a PIC using C is to study a simple program. The following example shows how to connect an LED to a PIC (PIC 16F877A used in your project) and flash it intermittently. The schematic for the project is shown in Figure below. Here an LED is connected to bit 0 of port B (i.e. pin RB0, pin36 on the chip) of a PIC16F877A microcontroller.

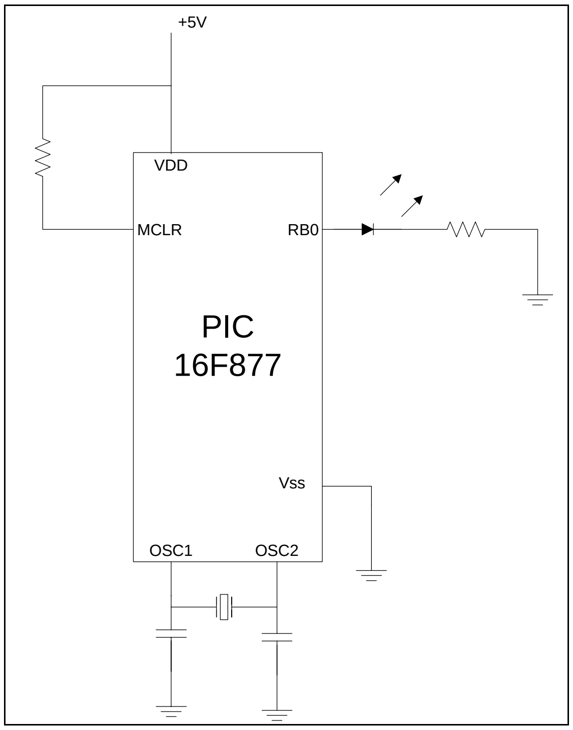


Fig 2 Schematic diagram of the LED flasher circuit

Hardware Setup

We will implement the above circuit on your DSX Experimenter board.

Note: We assume that you have already done the tests as described in the "Functional Testing Applications Manual" to make sure the Experimenter board is working and you are familiar with the board and how to program the chip.

To complete the above circuit you need to connect a single wire from RB0 (connection 0 on H5/PortB) to one of the LEDs (say connection 16 on H15) on the board. All the other components are pre-connected for you. See figure below. For this project you do not need any of the peripheral boards (such as the Stepper Motor Peripheral Module etc…)

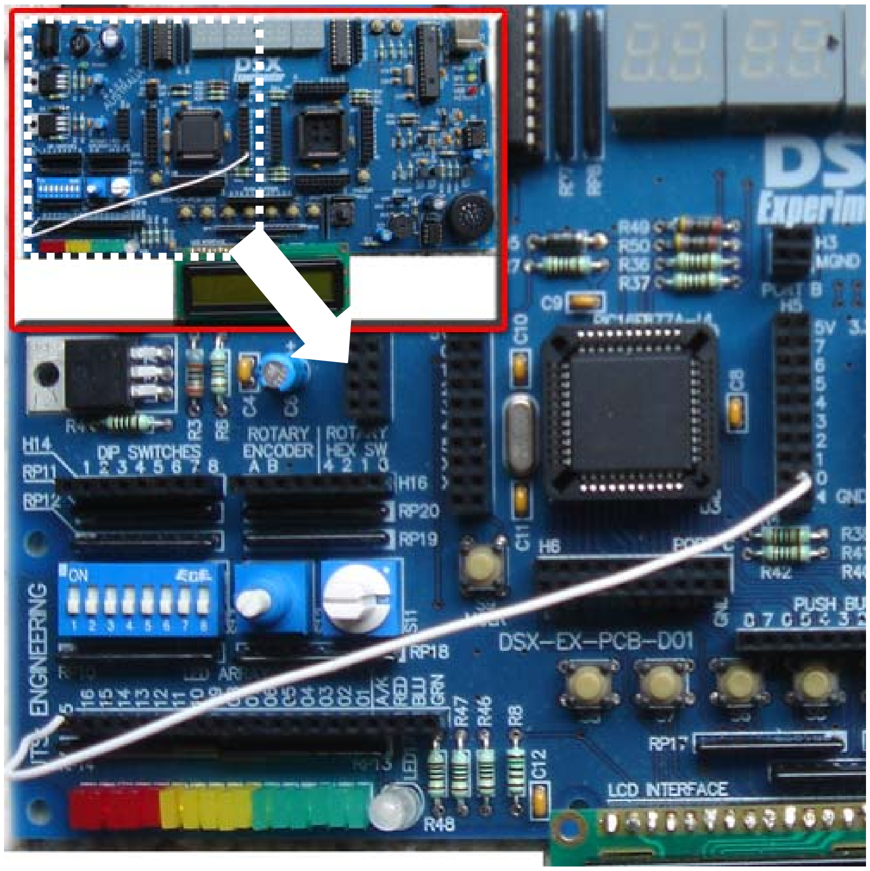


Fig 3 Actual wiring on the DSX board

Software Setup

There are many ways of writing code to achieve the goal. Here we will use a simple code which is easy to understand. What we try to do is to turn the bit RB0 on and off with a delay in-between so the LED connected to it turns on and off accordingly.

Follow the steps described below to program the chip with the code.

- Open the MPLAB IDE by double clicking on the "MPLAB IDE" icon.
- Run Project >> Project Wizard… to create a new project and follow the prompts, noting…
  - Step1: You are using "PIC16F877A" for device
  - Step2: Active Toolsuite: HI-TECH Universal ToolSuite
  - Step 3: Create New Project File: Using the "Browse" button navigate to where you want to create your new project and enter a name for the project (say flasher.mcp)
  - Step 4: Skip this step as you are yet to create the source files
  - Click "Finish" to complete the configuration
- Now create a new file by clicking File >> New
- Copy the following code into the new file.

```
#include <pic.h>
#include <delay.c>
__CONFIG(HS & WDTDIS & PWRTEN & BOREN & LVPDIS & DUNPROT & WRTEN & UNPROTECT);

void setup (void)
{
    TRISB = 0b00000000;
}

void main (void)
```

```
{
    setup();
    while (1)
    {
        RB0 = 1;
        DelayMs(250);
        DelayMs(250);
        RB0 = 0;
        DelayMs(250);
        DelayMs(250);
    }
}
```

- Save the file with the name “main.c” inside the project directory.
- Add this file to the project by clicking Project >> Add Files to Project… and selecting “main.c”
- You may be able to build this project at this stage by clicking the Project >> Build
    - Results of the build process are shown on the Output Window (under Build tab). If it says ********** Build failed! ************** for the following reason, “Error [141] …/main.c; 2.18 can't open include file "delay.c":”, then the compiler is not able to find the supporting file “delay.c” used in this example. You need to tell the compiler where to find this file. This file is part of the examples that ship with the PICC-Lite compiler and should be in the following (or similar) directory, “C:\Program Files\HI-TECH Software\PICC\lite\9.60\samples\delay”. To include this directory to the project search paths, click Project >> Build Options… >> Project and on the Directories tab under Show directories for: select Include Search Path, the click new button and enter the location of the “delay.c” folder.
    - Build the project.

PROGRAMMING THE CHIP AND TESTING

- Now you are ready to program the chip with your code. First power-up the DSX Experiment board and make sure the Power LED is turned on.
- Connect the USB cable to the board and to the host computer. Make sure that the necessary drivers are installed. (This should be the case if you have already done the functional tests for the DSX)
- To write the code to the chip we will use UTS DSX Kirra software specially designed for your DSX board.
    - Run the DSX Kirra program.
    - Make sure that "DSX Experimenter Module found and connected. PIC Device Found" is displayed.
    - You need to import the code to the program. For this click File >> Import Hex and navigate to your project folder created earlier. If you have successfully built the project, you should see a file called "flasher.hex". Select this file.
    - A message will be displayed as "Hex file successfully imported"
    - Now you can download the code to the flash memory of your microcontroller. For this simply click Write button on the PIC tab on DSX Kirra.
    - After a few seconds, the message "Programming Successful" will be displayed, and you should see the LED connected to RB0 start flashing!!!

UNDERSTANDING THE CODE

Now that you have a working code, let us try to understand the code in detail.

`#include <pic.h>` - This file is the standard include file for the PICC-Lite compiler. It contains the definitions for all the general purpose functions that can be directly accessed.

#include <delay.c> - This file is from the examples section installed by default when installing the PICC-Lite compiler. This file provides additional functionality to create execution delays in your code. Specifically, this file is needed to call the function "DelayMs(time)".

You can open this file in MPLAB to see how the code is implemented. The time parameter (in milliseconds) sets the delay length prior to executing the next statement in the code. You need to set the frequency of the crystal used in the board (ie. 20MHz for the DSX board) by adding the following line at the top of the "delay.c" file:

```
#define XTAL_FREQ     20MHZ
```

__CONFIG(HS & WDTDIS & PWRTEN & BOREN & LVPDIS & DUNPROT & WRTEN & UNPROTECT); - These are configuration options for the microcontroller. These need to be set appropriately. You can find details of these options on the PIC data sheet. Note that there are two underscores at the start of this instruction

void setup (void) { … } – This is a user defined function to set the appropriate i/o bits.

TRISB = 0b00000000; - This sets the Port B to be an output port.

void main (void){ … } – This is the main function which gets called at the beginning of the program execution.

setup(); - Here the user defined function setup() is called (discussed previously) to set Port B as an output port so we can connect the LED to one of the bits of this port.

while (1) { … } – Program enters an infinite loop in which the actual LED flasher code is enclosed. The loop is infinite because the logic expression (1) always evaluates to true

RB0 = 1; - Turn off the LED

DelayMs(250); - Wait for 250 ms (1/4 of a second). The maximum delay accepted by this function is 255 ms, so we have to call it two times to get a total ½ second delay.

RB0 = 0; - Turn on the LED

7.6 Handling User Input

In this second example, instead of flashing the LED we will make it toggle on/off by user input through a push button. The schematic for this is shown in the figure below.

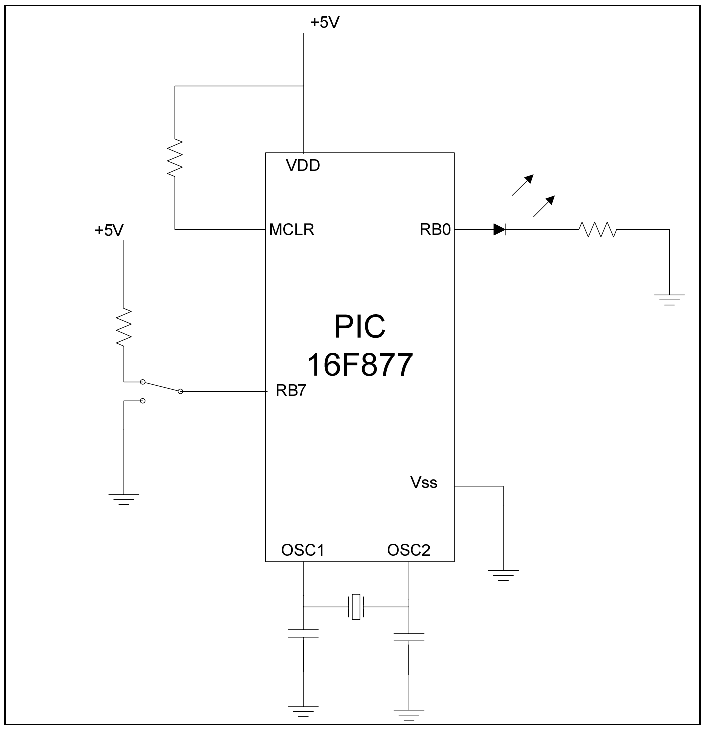


Fig 4 Schematic for connecting and LED and a switch

Here we are using two bits of PortB. Note how a switch is now connected to RB7, the most significant bit of PortB and as previously an LED is connected to RB0. You only need to connect another wire to the previous setup to complete this circuit. Connect a wire from RB7 (connection 7 on H5/PortB) to one of the pushbutton switches (say connection 8 on H17, ie S8) on the board.

As in the previous example, create a new project and copy the following code to a file called "main.c";

```
#include <pic.h>

#include <delay.c>
```

```
__CONFIG(HS & WDTDIS & PWRTEN & BOREN & LVPDIS & DUNPROT & WRTEN & UNPROTECT);

void setup (void)
{
    TRISB = 0b10000000;
    RB0 = 1;
}

void main (void)
{
    setup();
    while (1)
        {
            while (RB7 == 0);
            RB0 = !RB0;
            DelayMs(250);
            DelayMs(250);

        }
}
```

As previously, build this project and write it to the microcontroller using DSX Kirra. Test to see if the program is working correctly by pushing the switch (S7) several times. The LED should toggle its status every time the button is pressed and release.

Try to see if you could understand the code, especially the difference between this example and the previous one.

The important points note are,

TRISB = 0b10000000; - Note how the most significant bit(RB7) is set to 1 in port configuration. This tells the chip that RB7 should act as an input.

`while (RB7 == 1);` - The program waits inside this loop until the user press the button. (ie RB7)

`RB0 = !RB0;` - This toggles the state of the LED

7.7 USART Serial Communication

Finally we will try a simple code from the PICC-Lite example folder that illustrates serial port communication. To begin copy the contents of the PICC-Lite example usart (in a folder similar to "C:\Program Files\HI-TECH Software\PICC\LITE\9.60\samples\usart") to a new project folder.

Run MPLAB IDE and start the "Projec Wizard" as mentioned previously. Select the newly created folder as the project folder and add the files in the folder to the project. (there should be 3 files called "main.c", "usart.c" and usart.h").

Once the project is created, you will need to make a few changes for the code to be compatible with your setup.

- In the "main.c" add the following configuration settings near the top of the file as follows,

```
#include <stdio.h>
#include <pic.h>
#include "usart.h"
__CONFIG(HS & WDTDIS & PWRTEN & BOREN & LVPDIS & DUNPROT &
WRTEN & UNPROTECT & DEBUGDIS);
```

In the "usart.h" change the clock speed to match the DSX Experimenter board,

```
#define FOSC 20000000L
```

Build the project and program the chip with this code using DSX Kirra.

Testing Serial Communication

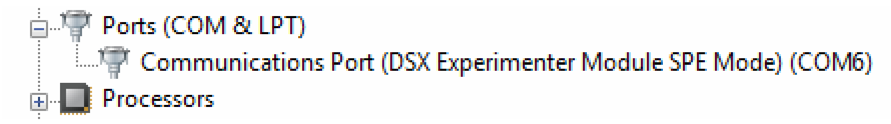
Note: We assume that you have already followed the DSX Experimenter Functional Testing Application Manual, to make sure that you are able to connect to the DSX Experimenter board through Serial Port Emulation (SPE), that you have already installed the driver software and HyperTerminal is configured with appropriate settings. (ie. Bits per second:9600, Data bits:8, Parity: None, Stop bits: 1 and Flow Control : NONE).

Once the chip has been programmed you need to change the OP MODE of the DSX Experimenter to SPE. Make sure that you have connected the USB cable to the PC and the Experimenter board and press the MODE button on the DSX Experimenter. The SPE LED will turn and will remain lit (if all is well).

Now start the Hyper Terminal (Usually in Start >> Programs >> Accessories >> Communications >> Hyper Terminal)

Open the DSX Experimenter Module Test (you have created this during the Functional Testing as set out in the Functional Testing Applications Manual) configuration.

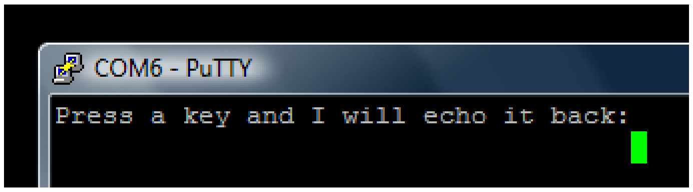
Note: If you run into issues connecting to the Experimenter board make sure that the configuration settings are correct (Bits per second:9600, Data bits:8, Parity: None, Stop bits: 1 and Flow Control : NONE) and you are connecting through the appropriate COM port. You can check the COM port number by Start >> Control Panel >> System >> Hardware Tab and clicking Device Manager. Expand Ports,



Ports (COM & LPT)
Communications Port (DSX Experimenter Module SPE Mode) (COM6)
Processors

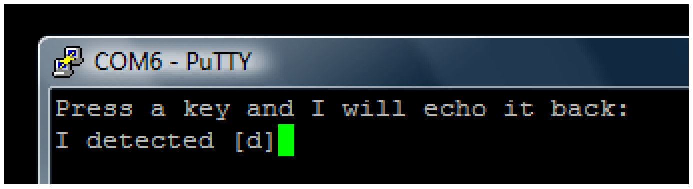
In the above example, DSX Experimenter SPE is connected as COM6.

Once the Hyper Terminal is connected to the Experimenter board, press S9/MCLR near the PIC on the DSX Experimenter board. Following message should be displayed on the Hyper Terminal,



```
Press a key and I will echo it back:
```

Now when you press a key the PIC will echo it back,



```
Press a key and I will echo it back:
I detected [d]
```

7.8 Accessing the Internal EEPROM

Hi-tech provides a number of inbuilt functions and macros which can be used to access memory locations in the internal EEPROM memory available on the PIC16F877A.

The __EEPROM_DATA( ); macro can be used to load values into the EEPROM when the hex file is loaded. To uses this macro you must supply 8 arguments which are loaded sequentially into the EEPROM memory starting from location 0. If you do not need all 8 memory locations you should write 0's for the remaining argument. If you need to preload more than 8 memory locations you should use multiple instances of the macro. Subsequent macro calls will continue writing EEPROM data from where the previous macro finished. Note that there are two underscores at the start of this macro.

Example:

```
__EEPROM_DATA(1,2,3,4,5,6,7,8);
__EEPROM_DATA(9,10,11,12,13,14,15,16);
```

The eeprom_read( ) function allows the contents of a EEPROM memory location to be read during the program execution. You must supply the memory address (char) as an argument when you call this function. The function returns the contents of the EEPROM memory.

Example:

```
temp = eeprom_read(0);
```

The eeprom_write( ) function allows data to be stored in the EEPROM during program execution. You must supply the memory address (char) followed by the write data (char) as arguments to the function.

Example:

```
unsigned char data = 10;
unsigned char address = 1;
eeprom_write(address, data);
```

Reference

In preparation of these notes several sources have been referenced extensively and the material is collectively acknowledged here. Reader is encouraged refer to these and many other text books and online material available in order to better understand the concepts and also to improve the efficiency and effectiveness of programs.

Microcontroller based applied digital control, Ibrahim Dogan, Chichester, England, Hoboken, NJ: John Wiley, c2006

Embedded C programming and the microchip PIC, Richard Barnett, Larry O'Cull, Sarah Cox. Clifton Park, NY : Thomson/Delmar Learning, c2004

PIC16F87X Data Sheet 28/40-Pin 8-Bit CMOS FLASH Microcontrollers, Microchip Technology Incorprated, c2001

DSX Functional Testing Applications Manual (Autmn 2008), DSX-FTAM-A08v1, Mark Benjamin, UTS: Engineering, CSP Group, c2008

DSX Quick Start Guide, DSX-QSG-01A, Mark Benjamin, UTS: Engineering, CSP Group, c2007