5 Lab 5 – Interrupts (part 1)

5.1 Aim

This laboratory practical will introduce you to the concept of interrupts, which are an important feature of all modern computer systems, including microcontrollers. This lab will enable you to comprehend the purpose of interrupts, and the way in which they can be utilised to optimise wasteful processes such as event polling loops. Consideration of the different hardware interrupts available on the PIC16F882 will be undertaken in this laboratory practical, and procedures will be developed for their proper configuration and servicing.

5.2 Learning Outcomes

- Comprehension of the concept of interrupts and have knowledge of the hardware interrupts available on the PIC16F882.
- Knowledge of how interrupts are implemented and serviced on the PIC16F882.
- Ability to develop code that utilises interrupts on the PIC16F882 to carry out useful tasks.

5.3 Background

5.3.1 Introduction to interrupts

Interrupts are a feature of every modern computer system, whether it be a complex multicore CPU in a desktop PC, or a simple 8 pin microcontroller in an electric toothbrush. Interrupts allow computer systems to react to *events* without using time wasteful processes such as polling loops. Examples of hardware events could be a logic state change on an external pin, a timer overflow, or the completion of an analogue-to-digital conversion.

5.3.2 Overview of instruction execution in a microcontroller

During normal code execution, the CPU retrieves instructions from the program memory location stored in the program counter. Under normal operation, after an instruction has been executed, a *program counter* automatically increments and the instruction held in the next address in the program memory is retrieved, see figure 5-1.

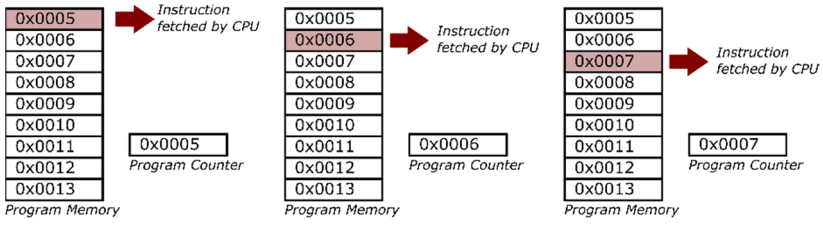


Figure 5-1 CPU instruction execution

In this way instructions are executed in a linear step-wise fashion, one after another. The *program counter* drives the retrieval and execution of instructions from the program memory. The program counter can also be loaded with specific memory addresses to cause the CPU execution to jump (*goto*) to the start of another sequence of instructions. This happens when branches, loops or function calls need to be implemented.

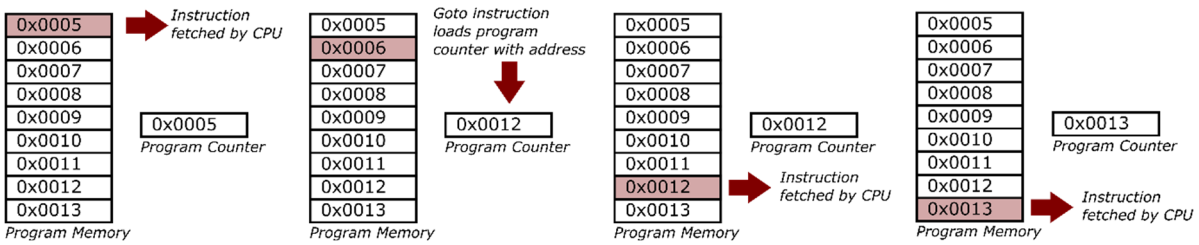


Figure 5-2 Action of a *goto* instruction on CPU instruction execution

A *subroutine* (known as a *function* in C/C++) call also causes the code to jump (*goto*) to the start of a specific set of instructions. The key feature of a subroutine is that once its instructions have been executed, the CPU returns back to the program memory address it was at prior to the initial subroutine call. This allows the same subroutine instructions to be called from multiple points in the programme execution because instruction execution will always return to the original memory location. To be able to provide this capability a storage entity called a *stack* is required that can store the original memory address. A stack is generally several levels deep and is an example of a *last-in-first-out* (LIFO) data structure. Several addresses may be *pushed* onto the stack, which may then be *popped* back off in reverse order. This action is illustrated in figure 5-3.

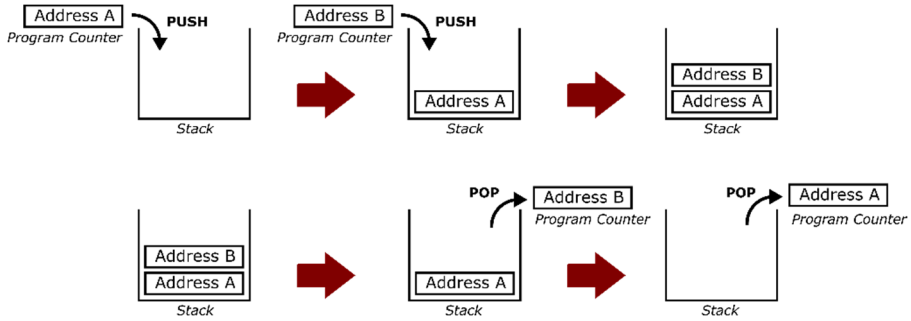


Figure 5-3 Pushing (top) and popping (bottom) addresses onto the stack

When a subroutine is called, the current program memory address on the program counter is *pushed* onto the stack. The program counter is then loaded with the program memory address containing the start of the subroutine instructions. Once the subroutine instructions have been executed, the subroutine *returns*. At this point, the program memory address stored in the *popped* from the stack and loaded onto the program counter. This action results in the CPU returning to the program memory location it was at prior to the function call. Figure 5-4 and figure 5-5 present a graphical overview of this process.

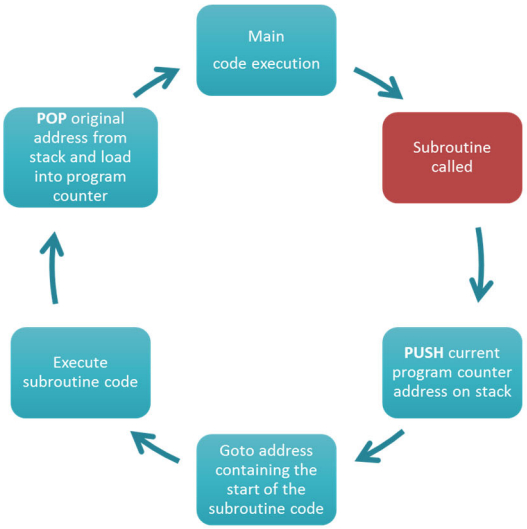


Figure 5-4 Overview of subroutine execution

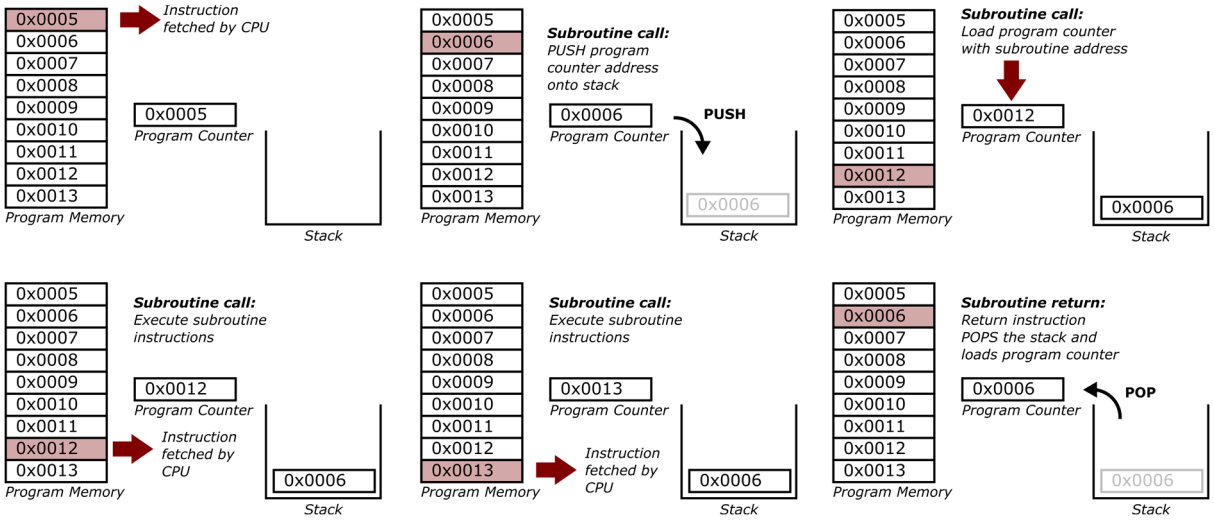


Figure 5-5 Detailed overview of subroutine execution including call and return

As described earlier, the stack has several levels which means several addresses may pushed onto it. This allows the subroutines to be called from within subroutines in a *nested* structure. Each subroutine call results in the pushing of the current program counter address onto the stack. When the function returns, the previous address is popped back off again. In the PIC16F882, the stack is only 8 levels deep so care must be taken not to nest too many subroutines (functions) or you may encounter a *stack overflow,* which will result in undefined behaviour. The XC8 compiler will produce warning during the build process if it detects the possibility of stack overflow occuring.

5.3.3 The interrupt service routine

The *interrupt service routine* (ISR) can be considered a special type of function that is automatically called when an interrupt event occurs. Figure 5-6 shows the cycle of actions that takes place when an interrupt event occurs. The sequence of events is precisely the same as that which occurs during a function call, except the interrupt vector is automatically loaded into the program counter by the hardware, which results in the ISR being called.

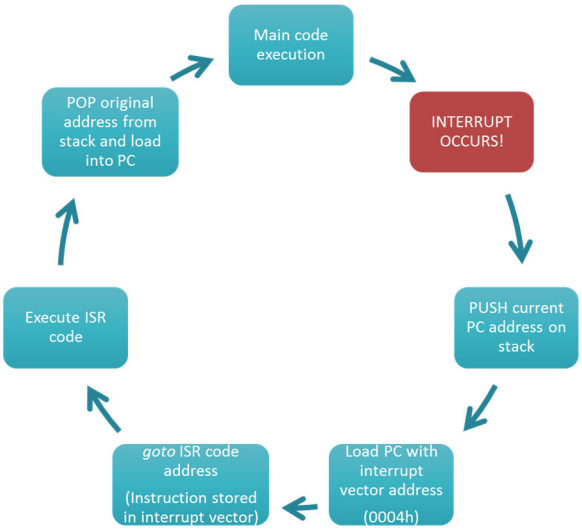


Figure 5-6 Overview of interrupt servicing

MCUs may have several interrupt vectors, each of which link to a different ISR. This allows the useful grouping of interrupts event types in more complex devices, for instance into different levels of importance or *priority.* The PIC16F882 is a relatively simple device having only 14 separate interrupt events. It has only one interrupt vector and thus only one possible ISR.

ISR code is often termed *background* code. When an interrupt occurs, the ISR code takes priority over the *foreground* code, which otherwise execute continously.

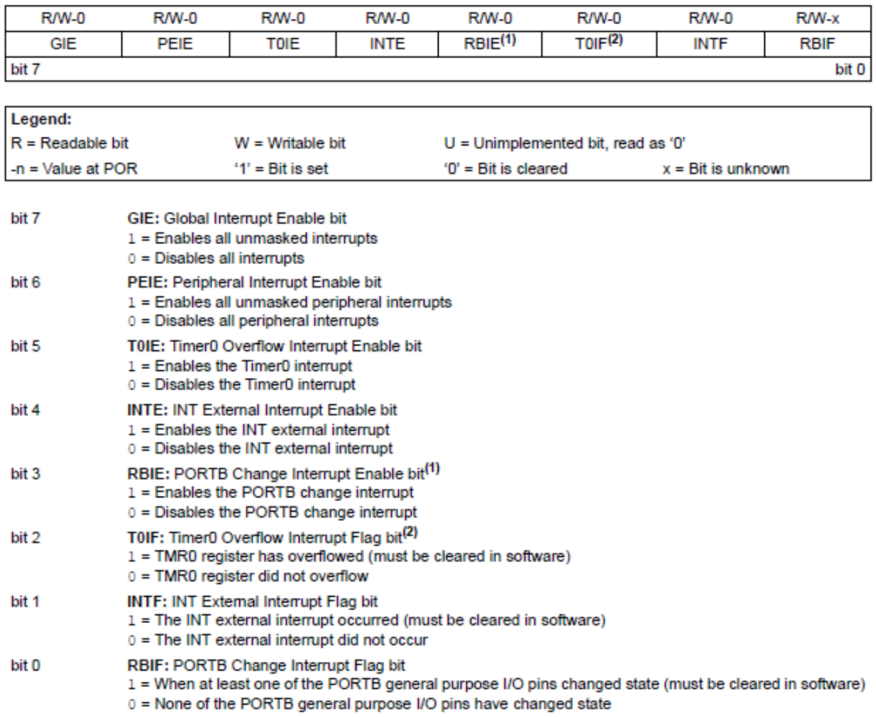
5.3.4 Interrupts on the PIC16F882

For an event to actually trigger an interrupt, the interrupt must first be enabled by setting configuration bits in the specific SFRs. In this laboratory practical we will be considering only three of the interrupts available on the PIC16F882 (see table 5-1).

Table 5-1 The 3 interrupts to be studied in this laboratory practical

Interrupt	Event	INTCON bits
TMR0 overflow	Timer0 overflow occurs	T0IE, T0IF
PORTB change interrupt	Logic level change on any of the PORTB pins RB0:7	RBIE, RBIF
External interrupt	Rising edge or falling edge on pin RB0.	INTE, INTF

These three interrupts can all be configured via the INTCON (interrupt control) SFR. The INTCON SFR configuration bits are summarised in figure 5-7. You will notice that there are two bits associated with each interrupt, an *enable bit* and a *flag bit.* In addition there are two further bits, *GIE* (Global Interrupt Enable) and PEIE (Peripheral Interrupt Enable). GIE acts as a 'master switch' and disables all interrupts when set to zero. PEIE enables the remaining secondary interrupts which we will study in a later laboratory.



R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	PEIE	T0IE	INTE	RBIE(1)	T0IF(2)	INTF	RBIF
bit 7							bit 0

Legend:

R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7 **GIE:** Global Interrupt Enable bit
1 = Enables all unmasked interrupts
0 = Disables all interrupts

bit 6 **PEIE:** Peripheral Interrupt Enable bit
1 = Enables all unmasked peripheral interrupts
0 = Disables all peripheral interrupts

bit 5 **T0IE:** Timer0 Overflow Interrupt Enable bit
1 = Enables the Timer0 interrupt
0 = Disables the Timer0 interrupt

bit 4 **INTE:** INT External Interrupt Enable bit
1 = Enables the INT external interrupt
0 = Disables the INT external interrupt

bit 3 **RBIE:** PORTB Change Interrupt Enable bit(1)
1 = Enables the PORTB change interrupt
0 = Disables the PORTB change interrupt

bit 2 **T0IF:** Timer0 Overflow Interrupt Flag bit(2)
1 = TMR0 register has overflowed (must be cleared in software)
0 = TMR0 register did not overflow

bit 1 **INTF:** INT External Interrupt Flag bit
1 = The INT external interrupt occurred (must be cleared in software)
0 = The INT external interrupt did not occur

bit 0 **RBIF:** PORTB Change Interrupt Flag bit
1 = When at least one of the PORTB general purpose I/O pins changed state (must be cleared in software)
0 = None of the PORTB general purpose I/O pins have changed state

Figure 5-7 The INTCON register (adapted from the PIC16F882 datasheet)

For an event to actually trigger an interrupt, the relevant interrupt enable bit in the INTCON SFR must be set. In addition the GIE bit must also be set. The interrupt flags should also be cleared as a matter of good practice during configuration so the MCU is in a known state.

5.3.5 Interrupt configuration in MPLAB X and the XC8 compiler

Configuring interrupts can be done by writing the appropriate value to the INTCON register. For instance, in order to setup the Timer0 overflow register it is necessary to set the GIE and TMR0IE registers. All other bits in the INTCON register should be cleared as shown below.

GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF
1	0	1	0	0	0	0	0

In this case, writing 0xA0 to the INTCON register configures the interrupt as required.

5.3.6 Coding ISRs in the MPLAB X and the XC8 compiler

The XC8 compiler uses the non-standard, compiler specific keyword *interrupt* to define the interrupt service routine (ISR). The ISR is defined in the same way as any other C function but it must be of type void and contain no parameters. Furthermore a function prototype is not required, because the ISR is never called by the rest of the program, it is only called when an interrupt occurs. The syntax for defining an interrupt is as follows:

```
void __interrupt() functionName (void)
{
    // ISR code goes here
}
```

Because the PIC16F882 has only a single interrupt vector, the first job of the ISR is to check which interrupt has actually occurred and act accordingly. This is easily done by simply checking the relevant interrupt flags as part of an if-else-if chain. In the following code, all relevant interrupt flags are checked. Code specific to each interrupt is only executed if the relevant interrupt flag is set.

```
void interrupt myISR(void)
{
    if(T0IF)
    {
           // Execute  Timer0 overflow interrupt specific code
           INTCONbits.T0IF = 0;        // Reset flag
           return;
    }

    if(INTF)
     {
           // Execute RB0 external interrupt specific code
           INTCONbits.INTF = 0;        // Reset flag
           return;
    }
}
```

Keep to this structure rigidly. It is essential that you remember to reset the relevant interrupt flag after you have executed the interrupt specific code. If this is not done then you will inadvertently retrigger the interrupt when you return from the ISR and your code execution will get locked up.

TIP!

Remember when an event occurs, the relevant interrupt flag is set regardless of whether the interrupt is enabled or not. In the Timer0 laboratory practical we used this facility detect a Timer0 overflow event by polling the T0IF flag.

5.3.7 Good practice when programming interrupts

ISRs can be difficult to debug because they are hardware driven and respond to asynchronous events. As such special care should be taken when designing code that makes use of interrupts to ensure against undefined behaviour.

- Only enable the interrupts you want to use.
- Ensure interrupt flags are cleared when you enable the interrupts.
- Keep ISRs short and simple. This makes them easier to debug and manage. It also reduces the likelihood of missing further interrupt events. Limit ISRs to simple tasks like incrementing counter variables or changing custom flags. These can then be acted upon accordingly by the foreground code.
- Use a strict if-else-if structure to test interrupt flags in your ISR. This reduces the likelihood of undefined behaviour in the event of an interrupt configuration error elsewhere.
- The if-else-if chain in your ISR should test interrupt flags in order of likelihood. The most commonly occurring interrupt should be at the top of the chain to increase code efficiency.

5.4 Procedure

5.4.1 Exercise 1 – Implementing a Timer0 overflow interrupt

1) Create a new project in the MPLAB X IDE, add an empty source file and populate it with the following code listing.

```
// Filename:     Lab5Ex1.c
// Version:      1.0
// Date:         <Insert current date>
// Author:       <Insert your name>
//
// Description: Looping of a short timer delay to create a longer one

// PIC16F882 Configuration Bit Settings
#pragma config FOSC = INTRC_NOCLKOUT // Oscillator Selection bits
#pragma config WDTE = OFF       // Watchdog Timer Enable bit
#pragma config PWRTE = OFF      // Power-up Timer Enable bit
#pragma config MCLRE = ON       // RE3/MCLR pin function select bit
#pragma config CP = OFF         // Code Protection bit
#pragma config CPD = OFF        // Data Code Protection bit
#pragma config BOREN = OFF      // Brown Out Reset Selection bits
#pragma config IESO = OFF       // Internal External Switchover bit
#pragma config FCMEN = OFF      // Fail-Safe Clock Monitor Enabled bit
#pragma config LVP = OFF        // Low Voltage Programming Enable bit
#pragma config BOR4V = BOR40V   // Brown-out Reset Selection bit
#pragma config WRT = OFF        // Flash Program Memory Self Write Enable bits

#include <xc.h>             // Required by compiler, PIC specific definitions

void main(void)
{
    ANSEL = 0x00;              // Set all pins to digital
    ANSELH = 0x00;
    TRISA = 0x00;               // Set PORTA all outputs
    PORTA = 0x00;               // Clear PORTA
    //Timer0 setup
    OPTION_REGbits.T0CS = 0;    // Set clock source to internal (timer mode)
    OPTION_REGbits.PSA = 0;     // Set prescaler to Timer 0
    OPTION_REGbits.PS = 3;      // Set prescaler bits to 011 for 1:16
    //Interrupt setup
    INTCONbits.T0IE = 1;        // Enable Timer0 overflow interrupt
    INTCONbits.T0IF = 0;        // Clear Timer0 overflow interrupt
    INTCONbits.GIE = 1;         // Enasble interrupts

    while(1);                   // Infinite do-nothing loop
}

void __interrupt() my_ISR(void)
{
    // Declare a count variable. The static type ensures the variable remains
    // in memory between subsequent functions calls. Ordinarily, a volatile
    // variable would be destroyed on return.

    static unsigned char count = 0;

    if(INTCONbits.T0IF) // Check if interrupt has occurred
    {
        TMR0 = 131;     // Preload Timer 0
        count++;        // Increment count variable
        if (count == 5) // If x counts have been reached, execute background task
```

```
        {
            PORTA++;    // Background task
            count = 0;  // Reset counter for next time
        }
        INTCONbits.T0IF = 0;    // Reset interrupt floag
        return;
    }
}
```

2) Study the code listing carefully and build the project. Upload the HEX file onto the PIC16F882 and observe the status of PORTA. Consider what is happening in terms of the interaction between the main code, which is stuck in an infinite do-nothing loop, and the ISR.

   Now modify the code so that the PORTA LEDs count up in binary at a rate of 1 Hz. In order to do this you will need to modify the ISR.

   If necessary, review the timer calculations you used to produce a one second delay in section 3.4.2 to allow you choose a suitable Timer0 preload value and determine how many interrupts to count.

   Do not simply place a *for* loop in the ISR to create the delay, the foreground code should continue to execute between ISR calls.

3) Write down a low-level flow chart showing the operation of the ISR ONLY.

5.4.2 Exercise 2 – Adding foreground code functionality

1) Create a new project and modify the code you developed in the previous exercise so that instead of the main() function doing nothing, it performs a useful task. This will run as 'foreground' code, while the ISR routines are 'background' code.

   In this case the task is display a flashing message on the LCD display. You should display your firstname and surname on the top and bottom rows of the LCD respectively. The text should flash on and off at 500 ms intervals.

   Don't forget to add the custom LCD library files to the project. You can create the required delays for the LCD routines using the __delay_ms() macro. Do not forget to insert the following statement in your code listing, near the top of the file,

```
#define _XTAL_FREQ 4000000
```

   This defines the MCU oscillator frequency which is needed by the __delay_ms() function to calculate the delay correctly. Failure to include it will lead to a compilation error.

5.4.3 Exercise 3 – Implementing pushbutton detection using PORTB interrupt-on-change

1) The PORTB interrupt-on-change (RBIE / RBIF) is useful to allow for the detection of logic level changes on any PORTB GPIO pin. In order to use this feature the relevant pin on PORTB must be set as an input and configured in digital mode.

   Enter the following code listing into a new project, study it carefully, then upload the HEX file to the PIC16F882:

```
// Filename:      Lab5Ex3.c
// Version:       1.0
// Date:          <Insert current date>
// Author:        <Insert your name>
//
// Description: Looping of a short timer delay to create a longer one

// PIC16F882 Configuration Bit Settings
#pragma config FOSC = INTRC_NOCLKOUT // Oscillator Selection bits
#pragma config WDTE = OFF       // Watchdog Timer Enable bit
#pragma config PWRTE = OFF      // Power-up Timer Enable bit
#pragma config MCLRE = ON       // RE3/MCLR pin function select bit
#pragma config CP = OFF         // Code Protection bit
#pragma config CPD = OFF        // Data Code Protection bit
#pragma config BOREN = OFF      // Brown Out Reset Selection bits
#pragma config IESO = OFF       // Internal External Switchover bit
#pragma config FCMEN = OFF      // Fail-Safe Clock Monitor Enabled bit
#pragma config LVP = OFF        // Low Voltage Programming Enable bit
#pragma config BOR4V = BOR40V   // Brown-out Reset Selection bit
#pragma config WRT = OFF        // Flash Program Memory Self Write Enable bits

#include <xc.h>             // Required by compiler, PIC specific definitions

void main(void)
{
    //GPIO setup
    ANSEL = 0x00;              // Set all pins to digital
   ANSELH = 0x00;
    TRISA = 0x00;               // Set PORTA all outputs
    PORTA = 0x00;               // Clear PORTA
    TRISB = 0x01;               // Set RB0 (SW0) as input
    PORTB = 0x00;
    //Timer0 setup
    OPTION_REGbits.T0CS = 0;    // Set clock source to internal (timer mode)
    OPTION_REGbits.PSA = 0;     // Set prescaler to Timer 0
    OPTION_REGbits.PS = 3;      // Set prescaler bits to 011 for 1:16
    //Interrupt setup
    INTCONbits.RBIE = 1;        // Enable Timer0 overflow interrupt
    INTCONbits.RBIF = 0;        // Clear Timer0 overflow interrupt
    IOCBbits.IOCB = 1;          // Enable RBIE on RB0 only
    INTCONbits.GIE = 1;         // Enable interrupts

    while(1);                   // Infinite loop
}

void __interrupt() my_ISR(void)
{
    if(INTCONbits.RBIF)
    {
        PORTA++;
        PORTBbits.RB0 = 0;  // Write PORTB to clear interrupt condition (mismatch)
        INTCONbits.RBIF = 0;// Clear interrupt flag
    }
}
```

2) Next modify the code to implement the following **additional** functionality using the PORTB interrupt-on-change (RBIE / RBIF):

- When SW1 is pressed the LEDs will countdown in binary. This should be done using the RBIE / RBIF interrupt,

- When SW1 is pressed all LEDs on the tutorial board will be cleared. This should be done using the RBIE / RBIF interrupt.

5.4.4 Exercise 4 – Servicing multiple interrupts

1) Create a new project and modify the code in you developed in the previous section to add in the following functionality:

   Foreground code should be added into the main() function to display your name in the same manner specified in the previous exercise. However, this time you should make use of the Timer0 overflow interrupt to control the timing of the LCD flashing.

5.5 Further reading

More useful information on implementing ISRs can be found in **section 5.9** of the MPLAB XC8 C Compiler User Guide available on Brightspace or at,

http://ww1.microchip.com/downloads/en/DeviceDoc/50002053F.pdf

More MCU specific information on interrupts can be found in **section 8** of the PICmicro Mid-Range MCU Family Reference Manual available on Unilearn or at,

http://ww1.microchip.com/downloads/en/DeviceDoc/33023a.pdf