INTRODUCTION

The following labs are designed for Microchip’s Explorer 8 Development Board. The Explorer 8 Development board supports 8/14/20/28/40-pin 8-bit PIC microcontrollers as well as up to 80-pin PIM-mounted devices. The lessons included in this document are developed for high pin count devices (28 pins and above). MPLAB X projects for these devices can be downloaded from the Microchip web site at www.microchip.com/explorer8.

This document comprises 12 lessons utilizing the different peripherals and features of 8-bit PIC® MCUs while demonstrating the different capabilities of the Explorer 8 Development Board. Each lesson contains a brief description of the lab, code snippets, and discussions to make you become easily acquainted with the different peripherals and registers of PIC® MCUs. These lessons also make use of the MPLAB Code Configurator (MCC), an easy-to-use plugin tool for MPLAB X IDE that you can use to generate codes for a more efficient use of the CPU and memory resources. All labs are written in C language and are compatible with the latest XC8 compilers.

LESSONS

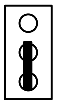
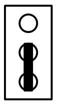
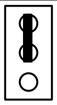
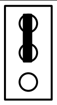
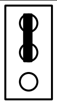
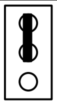
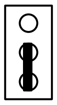
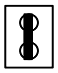
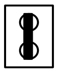
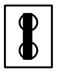
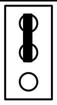
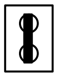
The lessons in this document are presented in the same order as they appear on the programmed labs. You can navigate through each lab by pressing the S2 button.

- **Lesson 1: Hello World (Turn On an LED)**
- **Lesson 2: Blink**
- **Lesson 3: Rotate (Moving the Light Across LEDs)**
- **Lesson 4: Analog-to-Digital Conversion (ADC)**
- **Lesson 5: Variable Speed Rotate**
- **Lesson 6: Debounce**
- **Lesson 7: Pulse-Width Modulation (PWM)**
- **Lesson 8: Timer1**
- **Lesson 9: Interrupts**
- **Lesson 10: Sleep/Wakeup**
- **Lesson 11: EEPROM(1)**
- **Lesson 12: High-Endurance Flash (HEF)(1)**

NOTE 1: These labs may not be applicable to all devices. Some devices have EEPROM only, HEF only, both, or none of the two. See your device datasheet for supported features.

JUMPER CONFIGURATIONS

Make sure to setup the following jumpers before demonstrating the labs.

JUMPER	CONFIGURATION	JUMPER	CONFIGURATION
J2 (External 9V DC Supply)	USB +5V / BRD +5V 	J37 (8 MHz External Osc)	RA5 / RA6 
J2 (USB-Powered)	USB +5V / BRD +5V 	J51 (ICSP programming)	RB7 / RA0 
J4 (RA5 as I/O pin)	RA5 / VCAP 	J52 (ICSP programming)	RB6 / RA1 
J14[1] (5V Supply)	+3.3V / +5V 	J59 (SPI Slave Select for LCD)	J59 
J21 (LEDs D8 through D5)	LED_B_EN 	J60 (LCD Reset)	LCD_RESET 
J36 (8 MHz External Osc)	RA7 / RA4 	J61 (LCD Power)	LCD_PWR 

Note 1: This setting is for 5V devices only which are mostly used in developing the labs. Devices such as the PIC18FxxK20 family should be supplied with 3.3V. Please refer to the specific device datasheet to prevent your device from being damaged.

INPUTS AND DISPLAY

- **Push Button Switch** – Two push button switches, S1 and S2, are provided on the board. S2 is used to switch to the next lab while S1 is needed to execute labs requiring external stimulus.
- **Potentiometer** – A 10kΩ potentiometer R25 is used in labs requiring analog inputs.
- **LEDs** - The Explorer 8 Development Board has eight blue LEDs (D8 through D1) that are connected to I/O ports RB3 through RB0 and RD3 through RD0, respectively. For the following labs, only LEDs D6, D7 and D8 are utilized to provide uniformity for both the 28-pin and 40-pin devices.

  NOTE: D5 lights up immediately when jumper J1 is connected because RB0 is always pulled high. RB0 is also the pin used for switch S1.
- **LCD** – A 16x2 Character LCD is used to display information/data regarding the currently running lab.

LESSON 1: HELLO WORLD (TURN ON AN LED)

1.1 Introduction

The first lesson shows how to turn on an LED.

1.2 Hardware Effects

LED D8 lights up and stays lit. The LCD displays “Hello World” on the first line and “LED_D8 = ON” on the second line.

1.3 Summary

The LEDs are connected to the microcontroller input-output pins (I/O). First, the I/O pin must be configured as an output. When one of these pins is driven high (RB3 = 1), the LED will turn on. The two logic levels ‘1’ (high) and ‘0’ (low) are derived from the power pins of the MCU. ‘1’ is equivalent to the device’s power pin (V_{DD}) and ‘0’ to the source (V_{SS}). V_{DD} may vary depending upon the supply requirements of the specific device used but V_{SS} is always connected to ground and is equal to ‘0’.

1.4 New Registers

Register	Purpose
LATx	Data latch
PORTx	Holds the status of all pins
TRISx	Determines if pins are input (1) or output (0)

LATx
The data latch (LATx registers) is useful for read-modify-write operations on the value that the I/O pins are driving. A write operation to the LATx register has the same effect as a write to the corresponding PORTx register. A read from the LATx register reads the values held in the I/O port latches.

PORTx
A read of the PORTx register reads the actual I/O pin value.

TRISx
This register specifies the data direction of each pin.

TRIS Value	Direction
1	Input
0	Output

An easy way to remember this is that the number ‘1’ looks like the letter ‘I’ for input and the number ‘0’ looks like the letter ‘O’ for output.

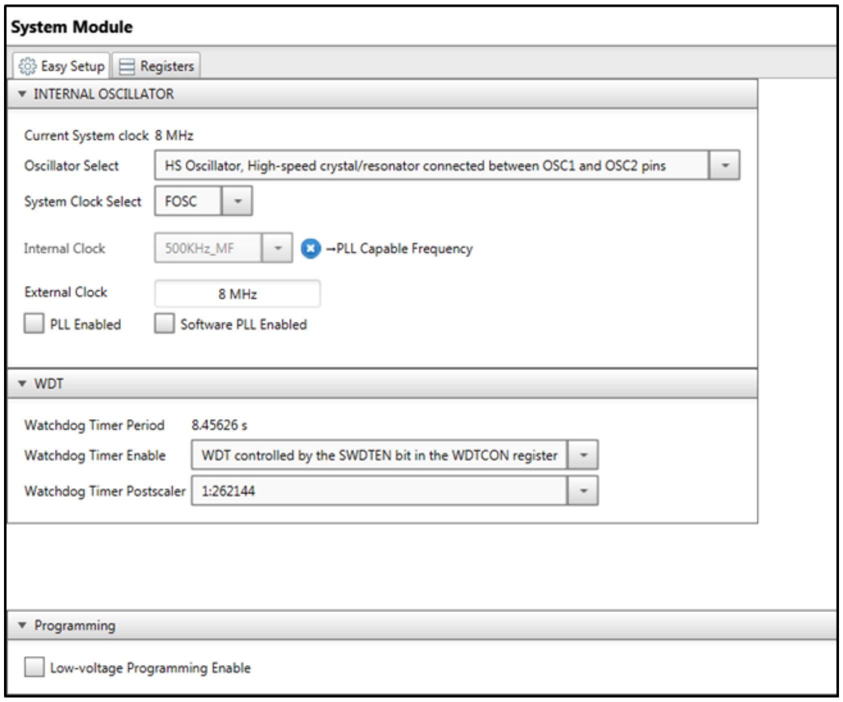
The user should always write to the LATx registers and read from the PORTx registers.

1.5 MCC Setup

1.5.1 System Setup

The initialization routine for all the labs mentioned in this document are configured using the MPLAB Code Configurator (MCC). The settings for the System module are shown in Figure 1-1.

FIGURE 1-1: MCC COMPOSER AREA– SYSTEM MODULE



Current System clock 8 MHz
Oscillator Select: HS Oscillator, High-speed crystal/resonator connected between OSC1 and OSC2 pins
System Clock Select: FOSC
Internal Clock: 500KHz_MF →PLL Capable Frequency
External Clock: 8 MHz
☐ PLL Enabled ☐ Software PLL Enabled

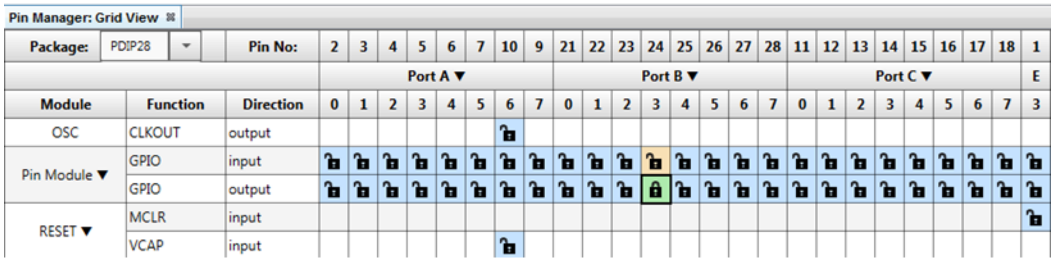
Watchdog Timer Period 8.45626 s
Watchdog Timer Enable: WDT controlled by the SWDTEN bit in the WDTCON register
Watchdog Timer Postscaler: 1:262144

☐ Low-voltage Programming Enable

1.5.2 Setting up the Pin as Output

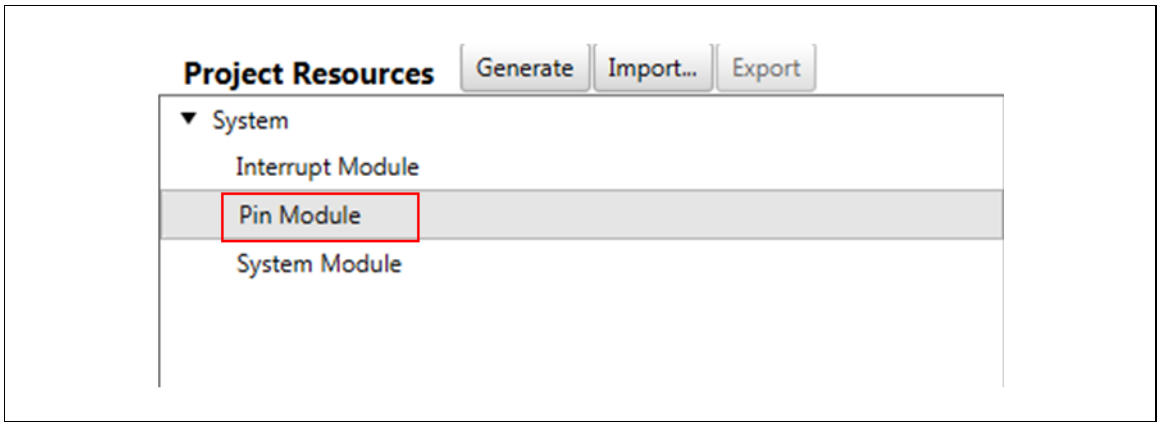
A GPIO pin can be set as input or output through the Pin Manager Window. RB3 is set as output as shown in Figure 1-2.

FIGURE 1-2: MCC PIN MANAGER WINDOW: GRID VIEW



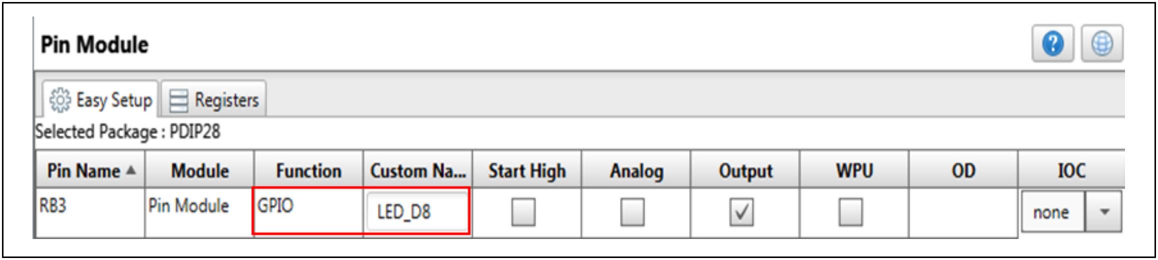
Package:	PDIP28	Pin No:	2	3	4	5	6	7	10	9	21	22	23	24	25	26	27	28	11	12	13	14	15	16	17	18	1
			Port A ▼								Port B ▼								Port C ▼								E
Module	Function	Direction	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	3
OSC	CLKOUT	output							🔓																		
Pin Module ▼	GPIO	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	GPIO	output	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔒	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
RESET ▼	MCLR	input																									🔓
	VCAP	input							🔓																		

FIGURE 1-3: PROJECT RESOURCES PANEL



In Figure 1-4, RB3 is provided with a custom name LED_D8.

FIGURE 1-4: MCC WINDOW – PIN MODULE

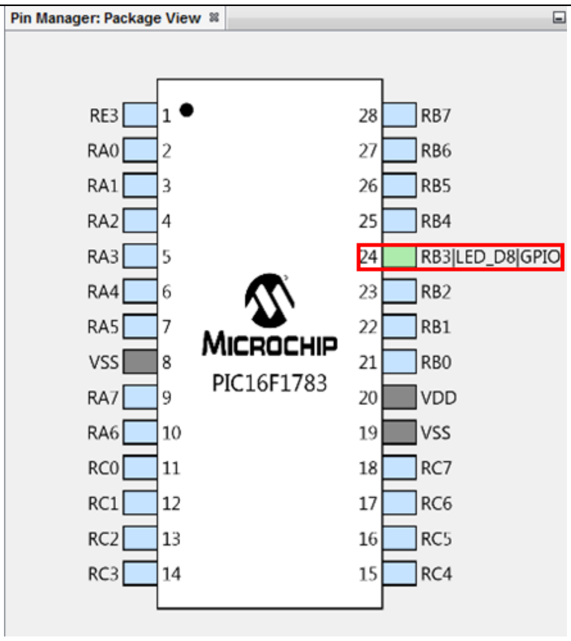


Selected Package : PDIP28

Pin Name ▲	Module	Function	Custom Na...	Start High	Analog	Output	WPU	OD	IOC
RB3	Pin Module	GPIO	LED_D8	☐	☐	✓	☐		none

The changes in the Pin Module also reflect on the Pin Manager: Package view. See Figure 1-5.

FIGURE 1-5: MCC PIN MANAGER WINDOW: PACKAGE VIEW



1.6 MCC Instructions

During code generation using the MPLAB Code Configurator, a `pin_manager.h` header file and a `pin_manager.c` source file are automatically created. `pin_manager.h` includes all the macro definitions and instructions for the different I/O pins (both analog and digital), whereas `pin_manager.c` includes the initialization code for these pins. Two of these macro instructions are used in this lab as shown below.

Instruction	Purpose
`LED_D8_SetHigh()`	Make the bit value of D8 (LATB3) a ‘1’ (V_{DD})
`LED_D8_SetLow()`	Make the bit value of D8 (LATB3) a ‘0’ (0V)

EXAMPLE 1.1: SETTING A BIT INTO ‘1’

```
LED_D8_SetHigh();
```

Before Instruction:

```
LATB3 = 0;
```

After Instruction:

```
LATB3 = 1;
```

EXAMPLE 1.2: SETTING A BIT INTO ‘0’

```
LED_D8_SetLow();
```

Before Instruction:

```
LATB3 = 1;
```

After Instruction:

```
LATB3 = 0;
```

1.7 C Language

A sample code written in C language for the “Hello World” lab is provided below.

EXAMPLE 1.3: C CODE FOR “HELLO WORLD” LAB

```
/**
  Section: Included Files
 */

#include "../../mcc_generated_files/pin_manager.h"
#include "../../labs.h"
#include "../../lcd.h"

/*
                             Application
 */
void HelloWorld(void){
        LEDs_SetLow();
        LCD_GoTo(0,0);
        LCD_WriteString("  Hello World   ");
```

```
    while(1){
        LED_D8_SetHigh();
        LCD_GoTo(1,0);
        LCD_WriteString("  LED_D8 = ON   ");
    }
}
```

```
#include "../../mcc_generated_files/pin_manager.h"
```

The header file `pin_manager.h` is generated automatically by the MPLAB Code Configurator (MCC). It provides implementations for pin APIs for all pins selected in the MCC GUI.

```
#include "../../labs.h"
```

This header file contains macro definitions, variable declarations, and function prototypes necessary for the project.

```
#include "../../lcd.h"
```

This header file contains function prototypes to drive the on-board LCD.

```
LCDGoto(0,0);
```

This function sets the cursor position to Row 0, Column 0 of the LCD. Row 0 corresponds to the first line of the LCD while Column 0 corresponds to the leftmost character. The LCD is a 16x2 character display which means that 16 Columns and 2 Rows are visible to the user.

```
LCD_WriteString ("  Hello World   ");
```

Send character string to the LCD to display the lab title. Spaces are used to display the text at the center.

```
LED_D8_SetHigh();
```

This statement calls the function `LED_D8_SetHigh()`. `LED_D8_SetHigh()` turns on LED D8.

```
LED_D8_SetHigh();
```

Equivalent:

```
#define LED_D8_SetHigh()    do { LATBbits. LATB3 = 1; } while(0)
```

This function is defined in `pin_manager.h` under the `MCC Generated Files` folder. It sets the LAT register of RB3 to `1`.

```
LCDGoto(1,0);
```

This function sets the LCD cursor position to the leftmost character of the second line by moving the cursor to Row 1, Column 0.

```
LCD_WriteString ("  LED_D8 = ON   ");
```

This function displays the status of LED D8 on the LCD.

LESSON 2: BLINK

2.1 Introduction

This lesson blinks the same LED used in **Lesson 1: Hello World** (LED_D8).

2.2 Hardware Effects

LED D8 blinks at a rate of approximately 1.5 seconds. The LCD displays "Blink" on the first line while displaying "LED_D8 = ON" and "LED_D8 = OFF" alternately every 1.5 seconds on the second line.

2.3 Summary

One way to create a delay is to spend time decrementing a value. In assembly, the timing can be accurately programmed since the user will have direct control on how the code is executed. In 'C', the compiler takes the 'C' and compiles it into assembly before creating the file to program to the actual PIC MCU (HEX file). Because of this, it is hard to predict exactly how many instructions it takes for a line of 'C' to execute. For a more accurate timing in 'C', this lab uses the MCU's TIMER1 module to produce the desired delay. TIMER1 is discussed in detail in **LESSON 8: TIMER1**.

2.4 MCC Setup

Setup procedure is the same found in **Section 1.5** of **LESSON 1: Hello World**.

2.5 New Registers

This lab utilizes Timer1 registers which will be discussed in **LESSON 8: TIMER1**.

2.6 MCC Instructions

Like the previous lab, this lab also uses an MCC-generated macro instruction which can be found in `pin_manager.h`.

Instruction	Purpose
`LED_D8_Toggle()`	Changes the bit value of D8 (LATB3) from '`0`' to '`1`', or '`1`' to '`0`'

EXAMPLE 2.1: TOGGLING A BIT

```
LED_D8_Toggle();
```

Before Instruction:

```
LATB3 = 0;
```

After Instruction:

```
LATB3 = 1;

Or
```

Before Instruction:

```
LATB3 = 1;
```

After Instruction:

```
LATB3 = 0;
```

2.7 C Language

A sample code written in C language for the “Blink” lab is provided below.

EXAMPLE 2.2: C CODE FOR “BLINK” LAB

```
/**
  Section: Included Files
 */
#include "../../mcc_generated_files/pin_manager.h"
#include "../../mcc_generated_files/tmr1.h"
#include "../../labs.h"
#include "../../lcd.h"

/**
  Section: Macro Declaration
 */
#define FLAG_COUNTER_MAX 6  // Maximum flag count to create 1.5 seconds delay

/**
  Section: Variable Declaration
 */
uint8_t flagCounter = 0;

/*
                             Application
 */
void Blink(void){
        LEDs_SetLow();
        LCD_GoTo(0,0);
        LCD_WriteString("     Blink      ");
        TMR1_StartTimer();
    while(1){
        if (LED_D8_PORT == HIGH){
            LCD_GoTo(1,0);
            LCD_WriteString("  LED_D8 = ON    ");
        }else{
            LCD_GoTo(1,0);
            LCD_WriteString("  LED_D8 = OFF   ");
        }
        while(!TMR1_HasOverflowOccured());
        TMR1IF = 0;
        TMR1_Reload();

        flagCounter++;
        if(flagCounter == FLAG_COUNTER_MAX){
            LED_D8_Toggle();
            flagCounter = 0;
        }
    }
}
```

```
//
```

This starts a comment. Any of the following text on this line is ignored by the compiler.

```
TMR1_StartTimer();
```

This function starts incrementing Timer1.

```
while(!TMR1_HasOverflowOccured());
TMR1IF = 0;
TMR1_Reload();
```

`TMR1_HasOverflowOccured()` is an MCC-generated Boolean routine to poll or to check for the Timer1 overflow flag. The program will first wait for the Timer1 flag (for approximately 250 ms) to be set before executing the next instructions, and will reload the same value of 250 ms to Timer1 (see **LESSON 8: Timer1**).

```
flagCounter++;
if(flagCounter == FLAG_COUNTER_MAX){
    LED_D8_Toggle();
    flagCounter = 0;
}
```

The variable ‘`flagCounter`’ increments every time `TMR1IF` is set until it reaches a value of ‘`6`’. This signifies that Timer1 has overflowed after 250 ms six times for a total of 1.5 seconds, before D8 is toggled. ‘`flagCounter`’ is then reset to ‘`0`’ and the process repeats.

```
LED_D8_Toggle();
```

Equivalent:

```
#define LED_D8_Toggle()    do { LATB3 = ~LATB3; } while(0)
```

This function is defined in `pin_manager.h` under the `MCC Generated Files` folder. It writes the complement of the previously written logic state on the RB3 PORT data latch (LATB3), making the pin “high” if previously “low” or vice versa.

LESSON 3: ROTATE (MOVING THE LIGHT ACROSS LEDS)

3.1 Introduction

This lesson would build on Lessons 1 and 2, which showed how to light up a LED and then make it blink using loops. This lesson incorporates three onboard LEDs (D6, D7 and D8) and the program will light each LED up in turn.

3.2 Hardware Effects

LEDs D6, D7 and D8 light up in turn every 500 milliseconds. Once D8 is lit, D6 lights up and the pattern repeats. The LCD displays “Rotate” on the first line and “LED_D*x* = ON” on the second line, where *x* indicates which LED is currently lit.

3.3 Summary

In C, we use Binary Left Shift and Right Shift Operators (<< and >>, respectively) to move bits around the registers. This lesson uses the left shift operator to move the light across the LEDs.

3.4 New Registers

There are no new registers for this lab. Refer to the previous labs for the registers used.

3.5 MCC Setup

Repeat setup procedure found in **Section 1.5** of **LESSON 1: Hello World** for LEDs D6 and D7.

3.6 MCC Instructions

This lesson uses the same MCC instructions found in **Section 1.6** of **LESSON 1: Hello World**.

3.7 C Language

A sample C code using binary shift operators is provided below.

EXAMPLE 3.1: SAMPLE C CODE FOR BINARY SHIFT OPERATORS

```
/**
  Section: Included Files
 */

#include "../../mcc_generated_files/pin_manager.h"
#include "../../labs.h"
#include "../../lcd.h"

/**
  Section: Local Variable Declarations
 */
uint8_t activeLedNum;


/*
                              Application
```

```
*/

void Rotate(void){
    // Begin with D6 high
    LEDs_SetLow();
    LED_D6_SetHigh();
    activeLedNum = 6;

    LCD_GoTo(0,0);
    LCD_WriteString("     Rotate     ");

    while(1){
        // Use delay to keep D6 ON for 0.5s
        __delay_ms(500);

        LEDs_ShiftRight();
        activeLedNum++;

        // If the last LED (D8) is lit, go back to the first LED (D6)
        if(activeLedNum > 8){
            LED_D6_SetHigh();
            activeLedNum = 6;
        }

        // Display which LED is ON on the LCD
        LCD_GoTo(1,0);
        LCD_WriteString("  LED_D  = ON   ");
        LCD_GoTo(1,7);
        LCD_WriteByte(activeLedNum+'0');
        }
}
```

```
LED_D6_SetHigh();
activeLedNum = 6;
```

Rotation starts by lighting up LED D6. ‘`activeLedNum`’ is a variable used to display on the LCD about what LED is currently lit. Here, the variable is equal to 6 which corresponds to D6 being lit up.

```
LEDs_ShiftRight();
```

Equivalent:

```
LATB <<= 1;
```

D8 through D6 are connected to PORTB<3:1>, respectively. Shifting 1 bit to the left turns off the currently lit LED and lights up the next LED to the right.

```
activeLedNum++;
if(activeLedNum > 8){
    LED_D6_SetHigh();
    activeLedNum = 6;
}
```

The ‘`activeLedNum`’ is being incremented every shift. After D8 is lit, D6 lights up again and the pattern repeats.

LESSON 4: ANALOG-TO-DIGITAL CONVERSION (ADC)

4.1 Introduction

This lesson shows how to configure the ADC, run a conversion, read the analog voltage controlled by the potentiometer (R25) on the board, and display the high order three bits on the display.

4.2 Hardware Effects

The top three MSBs of the ADC are displayed on the LEDs. Rotate the potentiometer to change the display. The LCD displays “ADC” on the first line and “ADC Result = *x*” on the second line where *x* shows the current ADC conversion result from 0 (`0b000`) to 7(`0b111`).

4.3 Summary

Most PIC® devices have on-board Analog-to-Digital Converters (ADC) or its enhanced version Analog-to-Digital Converter with Computation (ADC^2) with resolution ranging from 8 to 16 bits. The ADC voltage reference can either be internally generated or externally supplied and is usually selected through software. This lesson uses the device’s V_{DD} as the ADC voltage reference. The result from the ADC is represented by a ratio of the voltage to the reference (see Equation 4-1).

EQUATION 4-1: ADC RESULT

$$ADC = (V/V_{REF})*1023$$

Converting the ADC result back to voltage requires solving for V (see Equation 4-2).

EQUATION 4-2: CONVERTING ADC RESULT TO VOLTAGE

$$V = (ADC/1023)*V_{REF}$$

In this lesson, the PIC microcontroller’s ADC or ADCC module is configured using MCC (see Figure 4-1). ADCC functions at different operational modes but to mimic the normal ADC operation, it is set to basic mode. To configure the ADC, the following must be done:

1. Configure the ADC pin as an analog input.
2. Select ADC clock.
3. Select channel, result justification and VREF source.

4.4 New Registers

Register	Purpose
`ANSELx`	Determines if the pin is digital or analog.

ANSELx

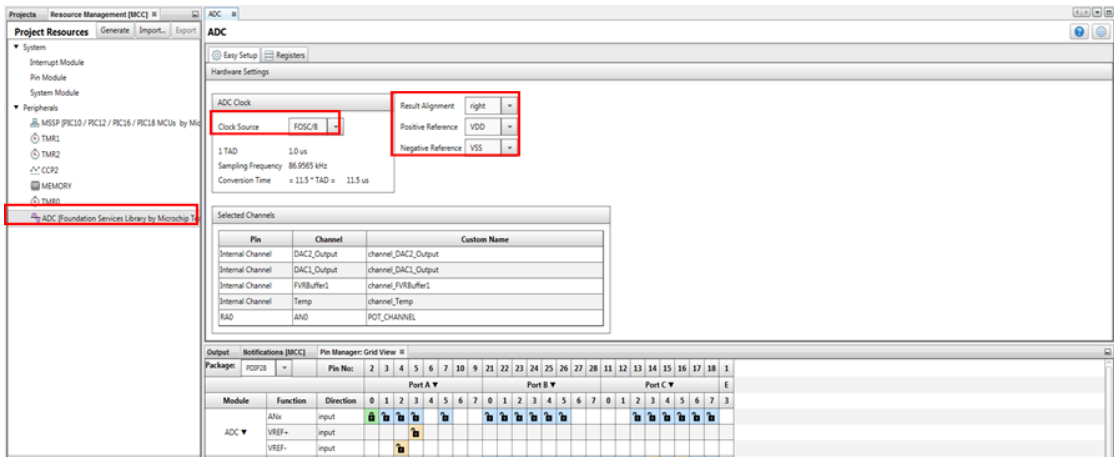
The ANSELx register determines whether the pin is a digital (1or 0) or analog (varying voltage). I/O pins configured as analog input have their digital input detectors disabled and therefore always read '0' and allow analog functions on the pin to operate correctly. The state of the ANSELx bits has no effect on digital output functions. When setting a pin to an analog input, the corresponding TRISx bit must be set to input mode to allow external control of the voltage on the pin.

This lesson sets RA0 as an analog input since the potentiometer (R25) will be used to vary the voltage.

4.5 MCC Setup

Select ADC from the Device Resources Panel and configure settings pertaining to the ADC clock and sampling frequency, result alignment and the ADC Channels to be used.

FIGURE 4-1: MCC WINDOW – ADC MODULE



ADC Clock

Clock Source: FOSC/8

1 TAD: 1.0 us

Sampling Frequency: 86.9565 kHz

Conversion Time: = 11.5 * TAD = 11.5 us

Result Alignment: right

Positive Reference: VDD

Negative Reference: VSS

Selected Channels

Pin	Channel	Custom Name
Internal Channel	DAC2_Output	channel_DAC2_Output
Internal Channel	DAC1_Output	channel_DAC1_Output
Internal Channel	FVRBuffer1	channel_FVRBuffer1
Internal Channel	Temp	channel_Temp
RA0	AN0	POT_CHANNEL

4.6 MCC Instructions

Instruction	Purpose
`ADC_Initialize()` or `ADCC_Initialize()`	Initialize the ADC module

`adc_result_t ADC_GetConversion(adc_channel_t channel)`
 	
 `uint16_t ADCC_GetConversion(adcc_channel_t channel)`
	This routine is used to select the desired channel for conversion and to get the result of ADC.

4.7 C Language

A sample code written in C language for the "ADC" lab is provided below.

EXAMPLE 4.1: C CODE FOR "ADC" LAB

```
/**
  Section: Included Files
 */

#include "../../mcc_generated_files/pin_manager.h"
#include "../../mcc_generated_files/adc.h"
#include "../../labs.h"
#include "../../lcd.h"

/**
  Section: Local Variable Declaration
 */
uint16_t adcResult = 0; // Used to store the result of the ADC

/*
                             Application
 */
void ADC(void){
    LEDs_SetLow();
    LCD_GoTo(0,0);
    LCD_WriteString("      ADC       ");

        while(1){
             // Get the top 3 MSBs and display these on the LEDs
             adcResult = ADC_GetConversion(POT_CHANNEL) >> 7;
             LEDs = adcResult << 1;

             // Display the ADC result on the LCD
             LCD_GoTo(1,0);
             LCD_WriteString(" ADC Result =   ");
             LCD_GoTo(1,14);
             LCD_WriteByte((adcResult)%10+'0');
        }
}
```

```
adcResult = ADC_GetConversion(POT_CHANNEL) >> 7;
```

ADC_GetConversion(POT_CHANNEL) Equivalent:

```
adc_result_t ADC_GetConversion(adc_channel_t channel){
    // select the A/D channel
    ADCON0bits.CHS = channel;

    // Turn on the ADC module
    ADCON0bits.ADON = 1;

    // Acquisition time delay
    __delay_us(ACQ_US_DELAY);

    // Start the conversion
    ADCON0bits.GO_nDONE = 1;
    // Wait for the conversion to finish
    while (ADCON0bits.GO_nDONE)
    {
    }
    // Conversion finished, return the result
    return ((ADRESH << 8) + ADRESL);
}
```

The function `ADC_GetConversion()` is generated automatically by the MCC. It selects the ADC channel, turns on the ADC module, sets up the delay, starts the conversion, and returns the result of the conversion. The result of the conversion is stored in the `'adcResult'` variable, which is defined as "unsigned 16-bit integer". For a right-justified 10-bit ADC, the 6 MSBs of `'adcResult'` will all be equal to '0'. Shifting the bits of `'adcResult'` to the right by 7 places leaves the 3 MSBs needed to display the result to the three LEDs (D6 to D8).

Note: Some devices with 12-bit ADCC cannot directly acquire conversion using `ADCC_GetConversion`. The data from `ADRESH` and `ADRESL` can also be acquired by using `ADCC_GetConversionResult` which will be shifted by 9 places to get 3 MSBs from 12-bit ADC.

The following shows how the top 3 MSBs are extracted from the result of the conversion in a right justified ADC.

Initialization:

`adcResult`

<15:8>								<7:0>							

After the initialization, `adcResult` is still empty and waiting for the conversion to be finished.

After conversion:

`adcResult`

ADRESH <15:8>								ADRESL <7:0>							
0	0	0	0	0	0	1	0	1	1	1	0	0	1	0	1

Once the conversion is done, the content of `ADRESH` and `ADRESL` are stored in `'adcResult'`. In this illustration, let's say that the value of `ADRESH` is `0b00000010` and `ADRESL` is `0b11100101`.

After shifting:

`adcResult`

<15:8>								<7:0>							
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

Shifting the value of `‘adcResult’` 7 places to the right leaves only the top 3 MSBs which is `0b101`.

```
LEDs = adcResult << 1;
```

`‘adcResult’` is shifted 1 place to the right to display the 3 MSBs on D8, D7 and D6 connected to PORTB<3:1>, respectively.

LESSON 5: VARIABLE SPEED ROTATE

5.1 Introduction

This lesson combines all of the previous lessons to produce a variable speed rotating LED display that is proportional to the ADC value. The ADC value and LED rotate speed are inversely proportional to each other.

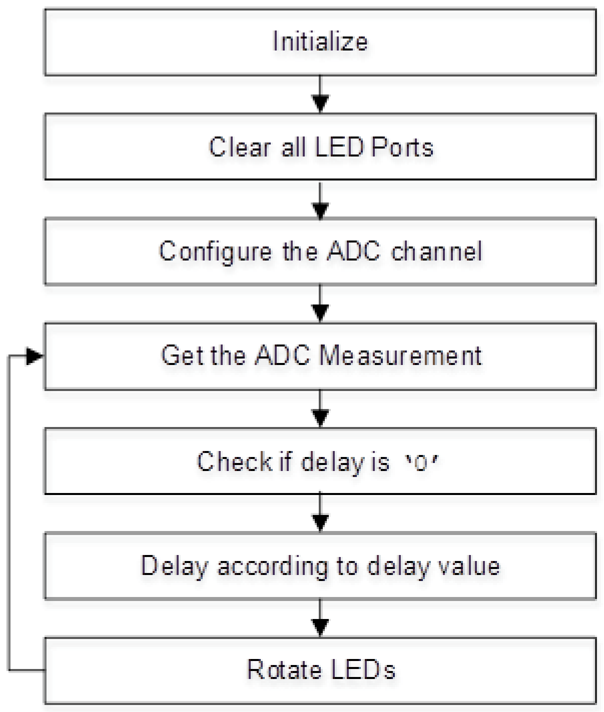
5.2 Hardware Effects

Rotate the potentiometer R25 counterclockwise to see the LEDs shift faster. The LCD displays “VSR” on the first line and “Delay = *xxx* ms” on the second line, where *xxx* indicates the time interval between LED shifts.

5.3 Summary

A crucial step in this lesson is to check if the delay value is 0. If it does not perform the zero check, and the ADC result is zero, the LEDs will rotate at an incorrect speed. This is an effect of the delay value underflowing from 0 to 255.

FIGURE 5-1: PROGRAM FLOW



5.4 New Registers

There are no new registers for this lab. Refer to the previous labs for the registers used.

5.5 MCC Setup

Use the setup procedures detailed in **Section 1.5** of **LESSON 1: Hello World** and **Section 4.5** of **LESSON 4: Analog-to-Digital Conversion (ADC)**.

5.6 MCC Instructions

This lesson uses the same MCC instructions with the previous labs.

5.7 C Language

A sample code written in C language for the "Variable Speed Rotate" lab is provided below.

EXAMPLE 5.1: C CODE FOR "VSR" LAB

```c
/**
  Section: Included Files
 */

#include "../../mcc_generated_files/pin_manager.h"
#include "../../mcc_generated_files/adc.h"
#include "../../labs.h"
#include "../../lcd.h"

/**
  Section: Macro Declaration
 */
#define ADDTL_DELAY         21

/**
  Section: Local Variable Declarations
 */
uint8_t activeLedNum;
uint8_t delay;
uint16_t totalDelay;

/*
                             Application
 */
void VSR(void){
   // Begin with D6 high
   LEDs_SetLow();
   LED_D6_SetHigh();
   activeLedNum = 6;

   LCD_GoTo(0,0);
   LCD_WriteString("      VSR       ");

   while(1){
        // Use the top 8 MSbs of the ADC result as delay
        delay = ADC_GetConversion(POT_CHANNEL) >> 2;
        // Compute for the total delay [1ms per delay value + 5ms add'l
        // delay + approx. 16ms for the other operations (e.g. LCD)

        totalDelay = delay + ADDTL_DELAY;
        __delay_ms(5);

        // Decrement the 8 MSbs of the ADC and delay each for 1ms
        while (delay-- != 0){
            __delay_ms(1);
        }

        LEDs_ShiftRight();
        activeLedNum++;

        // If the last LED (D8) is lit, go back to the first LED (D6)
        if(activeLedNum > 8){
            LED_D6_SetHigh();
            activeLedNum = 6;
        }

        // Display the total delay value in ms on the LCD
        LCD_GoTo(1,0);
        LCD_WriteString(" Delay =     ms ");
        LCD_GoTo(1,9);
        LCD_WriteByte(totalDelay/100+'0');
        LCD_WriteByte((totalDelay/10)%10+'0');
        LCD_WriteByte((totalDelay/1)% 10+'0');
    }
}
```

```
#define ADDTL_DELAY         21
```

The macro “`ADDTL_DELAY`” is used to add 21ms delay value which is computed by adding the minimum 5ms delay and approximately 16ms from other operations in the system.

```
delay = ADC_GetConversion(POT_CHANNEL) >> 2;
```

The 8 MSbs of the value resulting from the ADC is stored in the variable ‘`delay`’ which determines the speed of rotation.

```
__delay_ms(5);

while (delay-- != 0)
    __delay_ms(1);
```

A minimum delay of 5 ms is set, and then the ‘`delay`’ variable is decremented until it reaches ‘`0`’. At every decrement, another delay of 1 ms is set. This gives the rotation an additional delay in ms equal to ‘`delay`’. After the last decrement, the code for rotation is executed.

LESSON 6: DEBOUNCE

6.1 Introduction

Mechanical switches can be very noisy resulting to a rough electrical transition. As the contacts move against each other, the imperfections and impurities on the surfaces cause the electrical connection to be interrupted. The problem is commonly known as switch bounce. Some of the intermittent activity is due to the switch contacts bouncing off each other.

There are a variety of approaches in debouncing a switch so that a single press doesn’t appear like multiple presses. One of the simplest ways to switch debounce is to sample the switch until the signal is stable or continue to sample the signal until no more bounces are detected. The sampling time should be investigated depending upon the type and behavior of the switch. It is a good practice to debounce all switches in the system to eliminate mechanical switching glitches.

6.2 Hardware Effects

LEDs D6, D7 and D8 light up in turn on every S1 button press. Once D8 is lit and S1 is pressed, D6 lights up and the pattern repeats. Holding the button moves the light continuously across the LEDs. The LCD displays “Debounce” on the first line and “LED_D*x* = ON” on the second line, where *x* indicates which LED is currently lit.

6.3 Summary

This lesson uses a simple software delay routine to avoid the initial noise on the switch pin. The code will delay for only 120 ms, but should overcome most of the noise. The required delay amount differs with the switch being used. Some switches are worse than others.

6.4 New Registers

There are no new registers for this lab. Refer to the previous labs for the registers used.

6.5 MCC Setup

Repeat setup procedure found in **Section 1.5** of **LESSON 1: Hello World** for LEDs D6 and D7.

6.6 MCC Instructions

This lesson uses the same MCC instructions with the previous labs.

6.7 C Language

A sample code written in C language for the “Debounce” lab is provided below.

EXAMPLE 6.1: C CODE FOR “DEBOUNCE” LAB

```
/**
  Section: Included Files
 */

#include "../../mcc_generated_files/mcc.h"
#include "../../labs.h"
```

```
#include "../../lcd.h"


/**
  Section: Local Variable Declaration
 */
uint8_t activeLedNum;

/*
                             Application
 */
void Debounce(void){
        // Begin with D6 high
        LEDs_SetLow();
        LED_D6_SetHigh();
        activeLedNum = 6;

        LCD_GoTo(0,0);
        LCD_WriteString("    Debounce    ");
        LCD_GoTo(1,0);
        LCD_WriteString("    Press S1    ");

// The Switch is normally tied to VDD ...when it is pressed, the switch port
// is connected to GND
        if (SWITCH_S1_PORT == LOW){

            // Debounce by delaying
            for(uint8_t i=0; i<6; i++){
                __delay_ms(20);
            }

          // If S1 is still pressed, shift the light to the next LED to the
          // right
            if (SWITCH_S1_PORT == LOW){
                LEDs_ShiftRight();
                activeLedNum++;

                // If the last LED (D8) is lit, go back to the first LED
                //D6
                if(activeLedNum > 8){
                    LED_D6_SetHigh();
                    activeLedNum = 6;
                }

                // Display which LED is lit to the LCD
                LCD_GoTo(1,0);
                LCD_WriteString("  LED_D  = ON   ");
                LCD_GoTo(1,7);
                LCD_WriteByte(activeLedNum+'0');
            }
        }
}
```

```
if (SWITCH_S1_PORT == LOW){

      // Debounce by delaying
            for(uint8_t i=0; i<6; i++){
                __delay_ms(20);
            }
```

A 20 ms delay is looped 6 times to produce a total delay of approximately 120 ms. This delay is used to debounce switch S1.

```
        if (SWITCH_S1_PORT == LOW){
                 LEDs_ShiftRight();
                 activeLedNum++;

                 // If the last LED (D8) is lit, go back to the
                 // first LED (D6)
                 if(activeLedNum > 8){
                     LED_D6_SetHigh();
                     activeLedNum = 6;
                 }
```

This is almost similar with the Rotate lab instructions. The only difference is that `‘activeLedNum’` increments only after every press to move the light to the next LED.

LESSON 7: PULSE-WIDTH MODULATION (PWM)

7.1 Introduction

In this lesson, the PIC® MCU generates a PWM signal that lights an LED with an analog input controlling the brightness.

7.2 Hardware Effects

Rotating the potentiometer R25 will adjust the brightness of LED D8. The LCD displays “PWM” on the first line and “Duty Cycle = *xx*%” on the second line, where *xx* corresponds to the PWM duty cycle.

7.3 Summary

Pulse-Width Modulation (PWM) is a scheme that provides power to a load by switching quickly between fully on and fully off states. The PWM signal resembles a square wave where the high portion of the signal is considered the ON state and the low portion of the signal is considered the OFF state. The high portion, also known as the pulse width, can vary in time and is defined in steps. A longer ON time will illuminate the LED brighter. The frequency or period of the PWM does not change. The more steps applied, the longer the pulse width would be, thus supplying more power to the load. Conversely, decreasing the number of steps would shorten the pulse width and supply less power. The PWM period is defined as the duration of one cycle or the total amount of ON and OFF times combined.

This lesson utilizes either the Capture/Compare/PWM Module (CCP), or its enhanced version namely Enhanced CCP (ECCP) in PWM mode to dim an LED. It is recommended that the reader refer to the CCP/ECCP section of the data sheet to learn about the related registers. This lesson will briefly cover how to setup a single PWM.

The PWM period is specified by the PRx register. Timer2 increments TMR2 register which is compared to duty cycle register (CCPRxH:CCPRxL), using the internal comparator. CCPRxL is used to load CCPRxH. One can think of CCPRxL as a buffer which can be read or written to, as CCPRxH is a read-only register. When the value in the TMRx register is equal to the one in the PRx register, the following three events occur on the next increment cycle:

1. TMRx is cleared
2. The CCPx pin is set
3. The PWM duty cycle is latched from CCPRxL into CCPRxH

7.4 New Registers

Register	Purpose
CCPxCON	Sets the mode to be used
CCPRxL	Buffer for the 8 MSB of the duty cycle
CCPRxH	Holds the 8 MSB of the duty cycle
TxCON	Sets the Timer2/4/6 control settings
PRx	Sets the PWM period
TMRx	Timer2/4/6 register that increments from 00h on each clock edge and resets to 00h when its value matches with the PRx register.

CCP Module Registers

CCPxCON
The CCPxCON control register contains the bits needed to determine the mode to be used (Capture, Compare or PWM mode).

CCPRxL
CCPRxL contains the eight MSbs of the Duty Cycle and the DCxB bits of the CCPxCON register contain the two LSbs. This register can be written anytime during the period.

CCPRxH
The duty cycle value is not latched into CCPRxH until after the period completes (i.e., a match between PRx and TMRx registers occurs). While using the PWM, the CCPRxH register is read-only.

Timer2/4/6 Module Registers

The 'x' variable used in the following registers is used to designate Timer2, Timer4, or Timer6. For example, TxCON references T2CON, T4CON, or T6CON.

TxCON
TxCON contains the timer prescale and postscale bits as well as the TMRxON bit which starts incrementing TMRx when set to '1'.

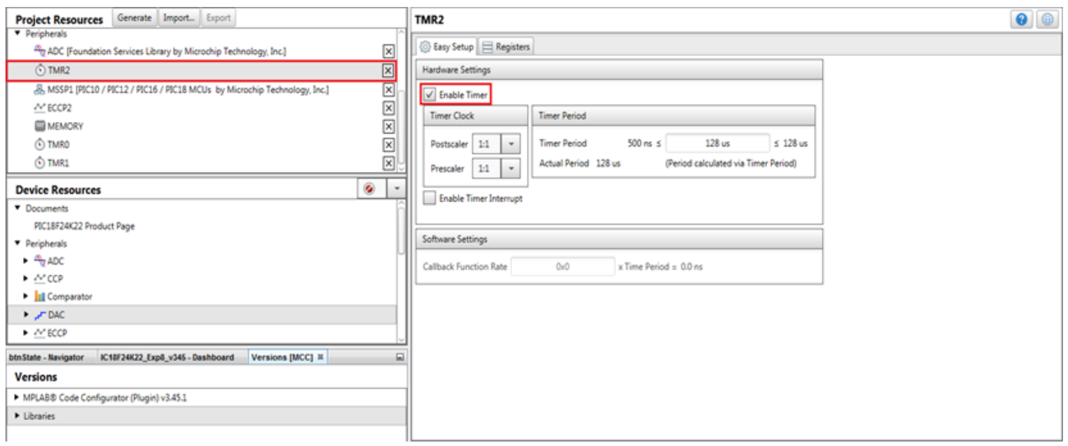
PRx
The PWM period is specified by the PR2 register.

TMRx
When TMRx is equal with PRx, TMRx is cleared, CCPx pin is set and CCPRxL is latched into CCPRxH.

7.5 MCC Setup

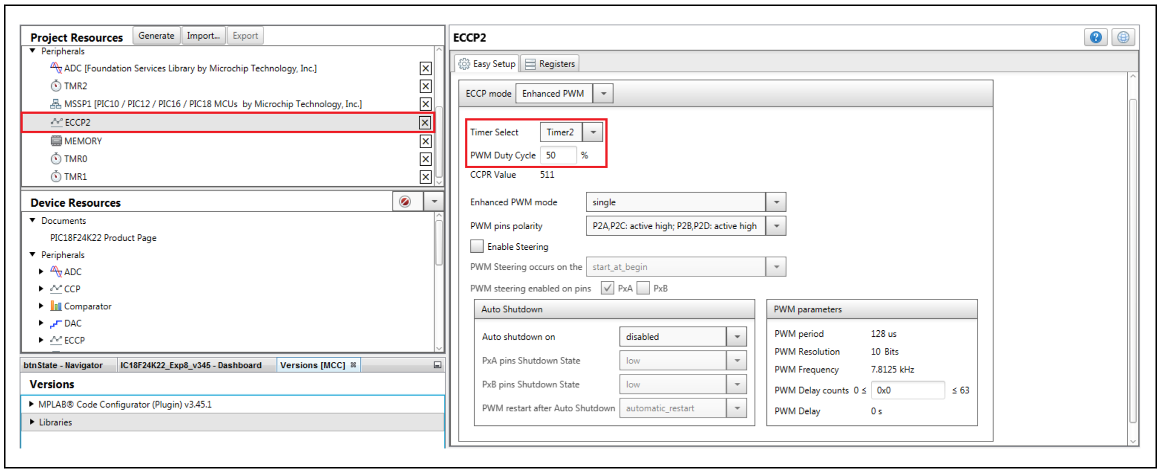
Shown below are the steps in configuring the ECCP module for standard PWM operation via MCC.

FIGURE 7-1: MCC WINDOW-- TIMER2 MODULE



Enable Timer [✓]
Postscaler 1:1
Prescaler 1:1
Timer Period 500 ns ≤ 128 us ≤ 128 us
Actual Period 128 us (Period calculated via Timer Period)
Enable Timer Interrupt [ ]
Callback Function Rate 0x0 x Time Period = 0.0 ns

FIGURE 7-2: MCC WINDOW – ECCP2: PWM MODULE



Peripherals
- ADC [Foundation Services Library by Microchip Technology, Inc.]
- TMR2
- MSSP1 [PIC10 / PIC12 / PIC16 / PIC18 MCUs by Microchip Technology, Inc.]
- ECCP2
- MEMORY
- TMR0
- TMR1

Documents
- PIC18F24K22 Product Page

Peripherals
- ADC
- CCP
- Comparator
- DAC
- ECCP

Versions
- MPLAB® Code Configurator (Plugin) v3.45.1
- Libraries

ECCP2

ECCP mode	Enhanced PWM
Timer Select	Timer2
PWM Duty Cycle	50 %
CCPR Value	511
Enhanced PWM mode	single
PWM pins polarity	P2A,P2C: active high; P2B,P2D: active high
Enable Steering	☐
PWM Steering occurs on the	start_at_begin
PWM steering enabled on pins	✓ PxA ☐ PxB

Auto Shutdown

Auto shutdown on	disabled
PxA pins Shutdown State	low
PxB pins Shutdown State	low
PWM restart after Auto Shutdown	automatic_restart

PWM parameters

PWM period	128 us
PWM Resolution	10 Bits
PWM Frequency	7.8125 kHz
PWM Delay counts 0 ≤	0x0 ≤ 63
PWM Delay	0 s

The PWM resolution defines the maximum number of steps that can be present in a single PWM period. A higher resolution allows for more precise control of the pulse width time and, in turn, the power that is applied to the load. In this lesson, the program will be using 10 bits of resolution.

EQUATION 7-1: PWM RESOLUTION

$$Resolution = \frac{log[4(PRx + 1)]}{log\ 2}\ bits$$

The term duty cycle describes the proportion of the ON time to the OFF time and is expressed in percentages, where 0% is fully off and 100% is fully on. The duty cycle is thus directly proportional to the power being supplied. The duty cycle is displayed on the board’s LCD.

Two conditions must hold true for this lesson:

1. 10 bits of resolution
2. No flicker in LED

7.6 MCC Instructions

This example uses the ECCP2 module in PWM mode and Timer2 as the PWM time base.

Instruction	Purpose
`TMR2_Initialize()`	Initialize Timer2
`TMR2_StartTimer()`	Start Timer2 by writing to TMRxON bit
`EPWM2_Initialize()`	Initialize the ECCP module in Enhanced PWM mode.
`EPWM2_LoadDutyValue(uint16_t dutyValue)`	Loads the Duty cycle value in the correct registers

7.7 C Language

A sample code written in C language for the “PWM” lab is provided below.

EXAMPLE 7.1: C CODE FOR “PWM” LAB

```
/**
  Section: Included Files
 */

#include "../../mcc_generated_files/pin_manager.h"
#include "../../mcc_generated_files/adc.h"
#include "../../mcc_generated_files/epwm2.h"
#include "../../mcc_generated_files/tmr2.h"
#include "../../labs.h"
#include "../../lcd.h"
/**
  Section: Local Function Prototypes
 */
void PWM_Output_D8_Enable (void);
void PWM_Output_D8_Disable (void);

/**
  Section: Local Variable Declarations
 */
uint16_t adcResult;
uint16_t dutyCycle;
/*
                             Application
 */
void PWM(void){
        LEDs_SetLow();
        TMR2_StartTimer();
        PWM_Output_D8_Enable();
        LCD_GoTo(0,0);
        LCD_WriteString("      PWM       ");

    while(1){
        adcResult = ADC_GetConversion(POT_CHANNEL);
        EPWM2_LoadDutyValue(adcResult);
        // Compute the duty cycle
        dutyCycle = (uint16_t)(((4*CCPR2L + CCP2CONbits.DC2B)*100UL)/(4*(PR2 + 1)));

        // Display the duty cycle on the LCD
        LCD_GoTo(1,0);
        LCD_WriteString((char*)"Duty Cycle =   %");
        LCD_GoTo(1,13);
        LCD_WriteByte((dutyCycle/10)%10+'0');
        LCD_WriteByte((dutyCycle/1)%10+'0');
   }
}
void PWM_Output_D8_Enable (void){
    // Set D8 as the output of CCP2 using the Alternate Pin Function Control
    // (APFCON)
    APFCONbits.CCP2SEL =1;
}
```

```
void PWM_Output_D8_Disable (void){
    // Restore D8 as a normal I/O pin
    APFCONbits.CCP2SEL = 0;
}
```

```
EPWM2_LoadDutyValue(adcResult);
```

Equivalent:

```
void EPWM2_LoadDutyValue(uint16_t dutyValue) {
    // Writing to 8 MSBs of pwm duty cycle in CCPRL register
    CCPR2L = ((dutyValue & 0x03FC) >> 2);

    // Writing to 2 LSBs of pwm duty cycle in CCPCON register
    CCP2CON = (CCP2CON & 0xCF) | ((dutyValue & 0x0003) << 4);
}
```

This function writes the 8 MSBs of `'adcResult'` to the CCPRxL register and the 2 LSBs to the DC2B<1:0> bits of the CCP2CON register. This 10-bit result is the PWM duty cycle.

Note: For devices that have Peripheral Pin Select instead of Alternate Pin Function, a PSS lock/unlock sequence is executed.

```
    // PPS Lock sequence
    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    // Unlock PPS
    PPSLOCKbits.PPSLOCKED = 0x00;

    // Set D8 as the output of CCP2
    RB3PPS = 0x0A;

    PPSLOCK = 0x55;
    PPSLOCK = 0xAA;
    // Lock PPS
    PPSLOCKbits.PPSLOCKED = 0x01;
```

This code is for the Peripheral Pin Select module that sets LED_D8 as an output for the PWM module.

LESSON 8: TIMER1

8.1 Introduction

This lesson will produce the same output as **Lesson 3: Rotate**. The only difference is that Timer1 will be used to provide the delay routine.

8.2 Hardware Effects

LEDs D6, D7 and D8 light in turn every 500 milliseconds. Once D8 is lit, D6 lights up and the pattern repeats. The LCD displays "TIMER1" on the first line and "Time = *xxx* secs" on the second line where *xxx* indicates a value between 0 and 100 corresponding to the time since TIMER1 started incrementing. A 1-second increment is equivalent to two LED shifts. When the display reaches 100, it will reset to 0 and the counting goes on.

8.3 Summary

Timer1 is a counter module which uses two 8-bit paired registers (`TMR1H:TMR1L`) to implement a 16-bit timer/counter in the processor. It may be used to count instruction cycles or external events that occur at or below the instruction cycle rate.

This lesson configures Timer1 to count instruction cycles and to set a flag when it rolls over. This frees up the processor to do meaningful work rather than wasting instruction cycles in a timing loop.

Using a counter provides a convenient and accurate method of measuring time or delay loops as it allows the processor to work on other tasks rather than counting instruction cycles.

8.4 New Registers

Register	Purpose
T1CON	Sets the timer enable, Prescaler, and clock source bits
TMR1H:TMR1L	16-bit timer/counter register pair
PIR1	Contains the Timer1 flag bit

T1CON
The Timer1 control register contains the bits needed to enable the timer, set-up the Prescaler and the clock source. `TMR1ON` turns the timer on or off. The `T1CKPS<1:0>` bits are used to set the Prescaler, while `TMR1CS<1:0>` bits select the clock source.

TMR1H:TMR1L
`TMR1H` and `TMR1L` are 8-bit registers that form a 16-bit timer/counter register pair. This timer/counter increments from a defined value until it reaches a value of `'255'` or `'0xFF'` each, and overflows. An overflow will set the Timer1 flag bit 'high' and trigger an interrupt when enabled.

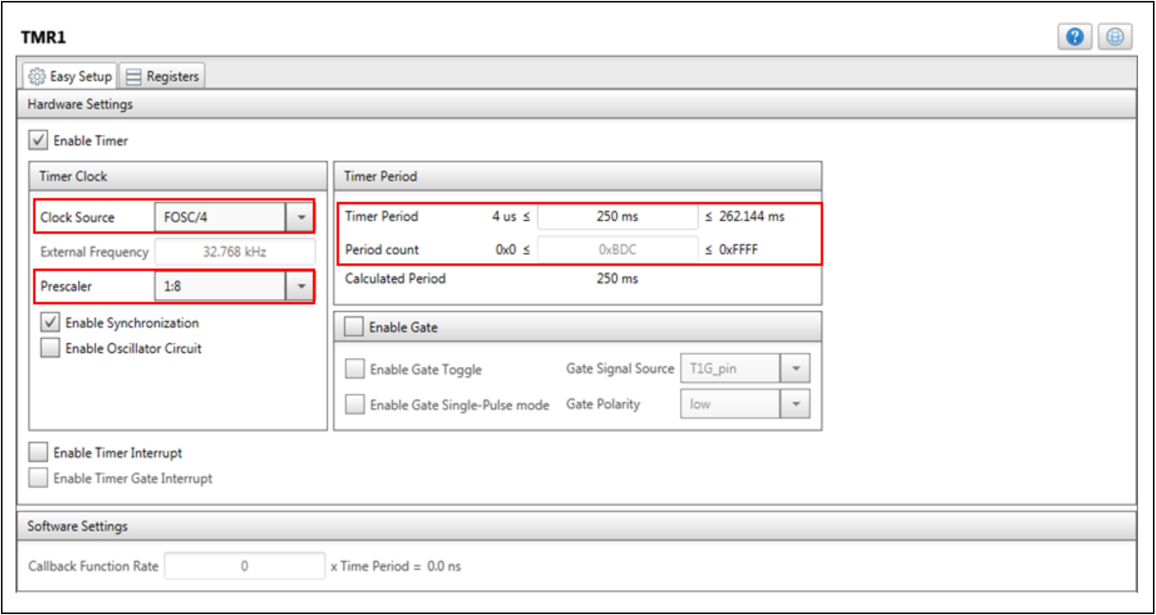
PIR1
This register contains `TMR1IF`, an interrupt flag that will be set to 'High' whenever Timer1 overflows.

8.5 MCC Setup

To make use of the TIMER1 module, select it from the Device Resources Panel under the Timer label. In this window, TIMER1 clock source, prescaler, period, gate enable and interrupt enable can be configured.

Shown in Figure 8-1 are the steps in configuring the Timer1 module via MCC.

FIGURE 8-1: MCC WINDOW – TIMER1 MODULE



Hardware Settings

✓ Enable Timer

Timer Clock

Clock Source	FOSC/4
External Frequency	32.768 kHz
Prescaler	1:8

✓ Enable Synchronization
☐ Enable Oscillator Circuit

Timer Period

Timer Period	4 us ≤	250 ms	≤ 262.144 ms
Period count	0x0 ≤	0xBDC	≤ 0xFFFF
Calculated Period		250 ms	

☐ Enable Gate

☐ Enable Gate Toggle — Gate Signal Source: T1G_pin
☐ Enable Gate Single-Pulse mode — Gate Polarity: low

☐ Enable Timer Interrupt
☐ Enable Timer Gate Interrupt

Software Settings

Callback Function Rate 0 x Time Period = 0.0 ns

8.6 MCC Instructions

Instruction	Purpose
`TMR1_Initialize()`	Initializes the TMR1
`TMR1_StartTimer()`	Starts the TMR1 operation
`TMR1_StopTimer()`	Stops the TMR1 operation
`TMR1_Reload()`	Reloads the TMR1 register

8.7 C Language

A sample code written in C language for the “Timer1” lab is provided below.

EXAMPLE 8.1: C CODE FOR “TIMER1” LAB

```
/**
  Section: Included Files
 */

#include "../../mcc_generated_files/pin_manager.h"
#include "../../mcc_generated_files/tmr1.h"
#include "../../labs.h"
#include "../../lcd.h"
/**
  Section: Macro Declarations
 */
#define FLAG_COUNTER_MAX    2        // 0.5 seconds delay
```

```
#define SHIFT_COUNTER_MAX   2       // 1 second delay
#define TIME_COUNT_MIN      0       // seconds
#define TIME_COUNT_MAX      100     // seconds

/**
  Section: Local Variable Declarations
 */
uint8_t flagCounter;
uint8_t timeCount;
uint8_t shiftCounter;
uint8_t activeLedNum;

/*
                             Application
 */
void Timer1(void){
        // Begin with D6 high
        LEDs_SetLow();
        LED_D6_SetHigh();
        activeLedNum = 6;

        LCD_GoTo(0,0);
        LCD_WriteString("    Timer 1    ");
        LCD_GoTo(1,0);
        LCD_WriteString("  Start Timer   ");

        flagCounter = 0;
        timeCount = 0;
        shiftCounter = 0;

        __delay_ms(2000);
        TMR1_StartTimer();

    while(1){
        while(!TMR1_HasOverflowOccured());
        // Clear the TMR1 interrupt flag
        TMR1IF = 0;
        // Reload the initial value of TMR1
        TMR1_Reload();
        // Increment the flagCounter
        flagCounter++;

        // If the second overflow occurs (0.5s delay)
        if(flagCounter == FLAG_COUNTER_MAX){
            LEDs_ShiftRight();
            activeLedNum++;

            // If the last LED (D8) is lit, go back to the first LED (D6)
            if(activeLedNum > 8){
                LED_D6_SetHigh();
                activeLedNum = 6;
            }

            flagCounter = 0;
            shiftCounter++;
        }
        // If the second shift occurs (1s count):
        if(shiftCounter == SHIFT_COUNTER_MAX){
            // Display the time count on the LCD
            LCD_GoTo(1,0);
            LCD_WriteString("Time =     secs ");
            LCD_GoTo(1,7);
            LCD_WriteByte(timeCount/100+'0');
            LCD_WriteByte((timeCount/10)%10+'0');
            LCD_WriteByte((timeCount/1)% 10+'0');

            shiftCounter = 0;
            timeCount++;        }

        if (timeCount > TIME_COUNT_MAX){
            timeCount = TIME_COUNT_MIN;
    }
}
```

```
if(flagCounter == FLAG_COUNTER_MAX){
            LEDs_ShiftRight();
            activeLedNum++;

            if(activeLedNum == 8){
                LED_D6_SetHigh();
                activeLedNum = 6;
            }

            flagCounter = 0;
            shiftCounter++;
}
```

The ‘`flagCounter`’ variable counts the number of Timer1 overflow. Timer1 is set to overflow every 250 ms and the LED light shifts to the right every two overflows (approximately 500ms).

```
if(shiftCounter == SHIFT_COUNTER_MAX){
    shiftCounter = 0;
    timeCount++;
}
```

On the other hand, ‘`shiftCounter`’ counts the how many times the light has shifted to the right. LED D6 is lit when the program starts. After the light has been shifted twice (from D6 to D7, and from D7 to D8), a total of two 500ms delay has elapsed. ‘`timeCount`’ is incremented to indicates a count of 1 second and its value will be displayed on the LCD.

```
if (timeCount > TIME_COUNT_MAX){
            timeCount = TIME_COUNT_MIN;
}
```

After ‘`timeCount`’ has reached its 100th second, it will reset to ‘0’ and the counting repeats.

LESSON 9: INTERRUPTS

9.1 Introduction

This lesson introduces interrupts and how they are useful. Here, an Interrupt on Change pin will be used to trigger an interrupt.

9.2 Hardware Effects

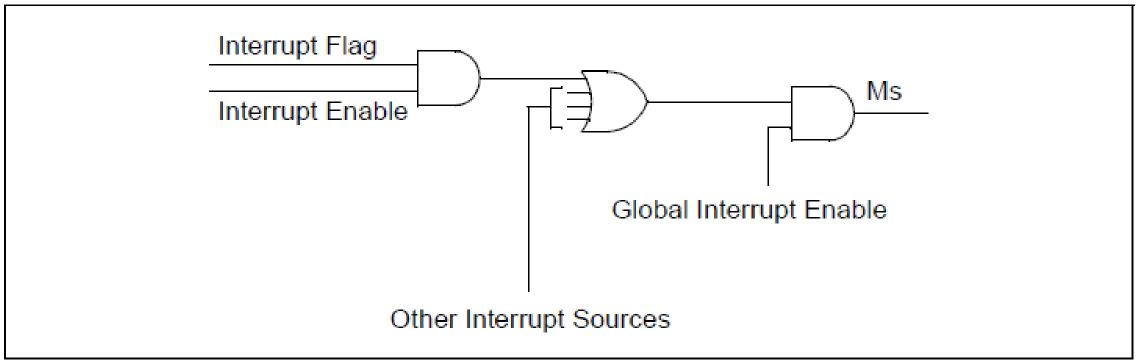
Light on LEDs D6, D7 and D8 rotate at a constant speed and pressing switch S1 reverses the direction of rotation. The LCD displays "Interrupt" on the first line and "Direction = *x*" on the second line, where *x* can either be "Left" or "Right" depending upon the direction of rotation.

9.3 Summary

The interrupt feature allows certain events to preempt normal program flow. This means that the microcontroller can be configured to be aware of its surroundings. Routines can be run upon occurrence of external events. The firmware should determine the source of the interrupt and act accordingly. All interrupts can be configured to wake the MCU from Sleep mode.

Most of the peripherals can generate an interrupt. Some of the I/O pins may be configured to generate an interrupt when they change state. When a peripheral needs service, it sets its interrupt flag. Each interrupt flag is ANDed with its enable bit and then these are ORed together to form a master interrupt. This master interrupt is ANDed with the Global Interrupt Enable (GIE). The enable bits allow the PIC microcontroller to limit the interrupt sources to certain peripherals. See the Interrupt Logic Figure in the PIC microcontroller data sheet for a drawing of the interrupt logic. Below is a simplified diagram.

FIGURE 9-1: INTERRUPT LOGIC



The PIC18 has a slightly different structure to accommodate interrupt priority. The enhanced mid-range core has only one interrupt vector. This means that whenever an interrupt occurs, the program counter goes to the interrupt service address, specifically address 0x0004. The PIC18 allows most interrupt sources to be assigned a high or low priority level. The high priority vector is at 0x0008 and the low at 0x0018. A high priority interrupt event will interrupt a low priority that may be in progress. This lesson will not utilize priority interrupts and will instead make use of the mid-range compatibility feature by clearing the IPEN bit. Both devices will now service from only one vector.

When an interrupt is being serviced, the Global Interrupt Enable (GIE) bit is cleared to disable further interrupts. The return address is pushed onto the stack and the PC is loaded with the interrupt vector address. Both the enhanced mid-range and PIC18 devices perform automatic context saving for the WREG, STATUS, and BSR registers. The FSR and PCLATH registers are saved only in the enhanced mid-range devices

The firmware within the Interrupt Service Routine (ISR) should determine the source of the interrupt by polling the interrupt flag bits. The serviced interrupt flag bits must be cleared before exiting the ISR to avoid repeated interrupts. Because the GIE bit is cleared, any interrupt that occurs while executing the ISR will be recorded through its interrupt flag, but will not cause the processor to redirect to the interrupt vector until the `retfie` instruction is executed, thereby enabling the GIE bit

NOTE: The example shown in the next sections implements a GPIO with Interrrupt-on-Change (IOC) capabilities. In some devices, the RB0 pin tied to S1 is also the INT0 pin by default. Like the IOC pins, the INT0 pin is also triggered by external interrupt sources.

9.4 New Registers

Register	Purpose
INTCON	Contains enable and flag bits for Global, Peripheral and other often used interrupts.
IOCxP	Contains associated pin for detecting positive edge for generating interrupt.
IOCxN	Contains associated pin for detecting negative edge for generating interrupt.
IOCxFx	Contains flag bits for Interrupt on Change pins.

INTCON

The INTCON register contains the enable bits for Global and Peripheral Interrupts, INT0, Interrupt-On-Change and Timer0. Flag bits for the latter three are also in this register.

IOCxP

The IOCxP register contains the enable bit of a pin to detect a rising edge. The flag bits associated in IOCxFx and IOCIF flag will be set upon detecting an edge.

IOCxN

The IOCxN register contains the enable bit of a pin to detect a rising edge. The flag bits associated in IOCxFx and IOCIF flag will be set upon detecting an edge.

IOCxFx

The IOCxFx register contains the associated flag bits of a pin when a change is detected.

9.5 MCC Instructions

Instruction	Purpose
`INTERRUPT_GlobalInterruptEnable()`	Enables Global Interrupts by setting the GIE bit.
`INTERRUPT_PeripheralInterruptEnable()`	Enables Peripheral Interrupts by setting the PEIE bit.

`INTERRUPT_GlobalInterruptDisable()`	Disables Global Interrupts by clearing the GIE bit
`INTERRUPT_PeripheralInterruptDisable()`	Disables Peripheral Interrupts by clearing the PEIE bit

9.6 C Language

Sample code written in C for interrupts and ISRs are shown below.

For PIC16F devices:

Main Program and Set-up

```
/**
  Section: Included Files
 */

#include "../../mcc_generated_files/pin_manager.h"
#include "../../mcc_generated_files/interrupt_manager.h"
#include "../../mcc_generated_files/tmr0.h"
#include "../../labs.h"
#include "../../lcd.h"
/**
  Section: Local Function Prototypes
 */
void Lab_Timer0_ISR(void);
void Lab_IOC_ISR(void);

/**
  Section: Local Variable Declaration
 */
uint8_t rotateDirection;
uint8_t rotateCounter;

/*
                             Application
 */
void Interrupt(void){
    LEDs_SetLow();
    // Begin with LED_D6 High
    LED_D6_SetHigh();

    INTERRUPT_GlobalInterruptEnable();
    INTERRUPT_PeripheralInterruptEnable();

    INTERRUPT_InterruptOnChangeEnable();
    INTERRUPT_IOCPositiveEnable();
    INTERRUPT_TMR0InterruptEnable();


    TMR0_SetInterruptHandler(Lab_Timer0_ISR);
    // Allows selecting an interrupt handler for IOCBF0 at application
    //  runtime
    IOCBF0_SetInterruptHandler(Lab_IOC_ISR);
    // Start the rotation from left to right
    rotateDirection = RIGHT;
    rotateCounter = 6;

    // Display the lab title and initial direction on the LCD
    LCD_GoTo(0,0);
    LCD_WriteString("   Interrupt    ");
    LCD_GoTo(1,0);
    LCD_WriteString("Direction: Right");
    //Wait for an interrupt to occur
    while(1);
}

void Lab_Timer0_ISR(void){
    if(rotateDirection == RIGHT){
        LEDs_ShiftRight();
        rotateCounter++;
        // If the last LED (D8) is lit, go back to the first LED (D6)
        if(rotateCounter > 8){
            LED_D6_SetHigh();
            rotateCounter = 6;
        }
    }
    else if(rotateDirection == LEFT){
        LEDs_ShiftLeft();
        rotateCounter--;
        // If the last LED (D6) is lit, go back to the first LED (D8)
        if(rotateCounter < 6){
            LED_D8_SetHigh();
            rotateCounter = 8;
        }
    }
}

void Lab_IOC_ISR(void){

    // Toggle the direction of the rotation
    rotateDirection ^= 1;

    if(rotateDirection == RIGHT){
        LCD_GoTo(1,0);
        LCD_WriteString("Direction: Right");
    }
    else if(rotateDirection == LEFT){
        LCD_GoTo(1,0);
        LCD_WriteString("Direction: Left ");
    }
}
```

```
INTERRUPT_GlobalInterruptEnable();
INTERRUPT_PeripheralInterruptEnable();
```

These are MCC-defined functions that enable the Global and Peripheral Interrupts, respectively. This is equivalent to setting the GIE and PEIE bits in the INTCON register. The user should always disable these functions before transferring to labs which do not require interrupt events.

```
INTERRUPT_InterruptOnChangeEnable();
```

Equivalent:

```
INTCONbits.IOCIE = 1;
```

Enabling Interrupt-on-Change (IOC) for specific R*xx* pins allows a device to be interrupted by a change the R*xx* pin’s state. One has the option to use either the positive or negative edge as the trigger.

```
INTERRUPT_IOCPositiveEnable();
```

Equivalent:

```
IOCBP0 = 1;
```

In this case, S1 (RB0) is set to trigger an interrupt at positive edge, or a change of low (0) to high (1). These can be implemented through manual coding or through MCC (see Figure 9-2).

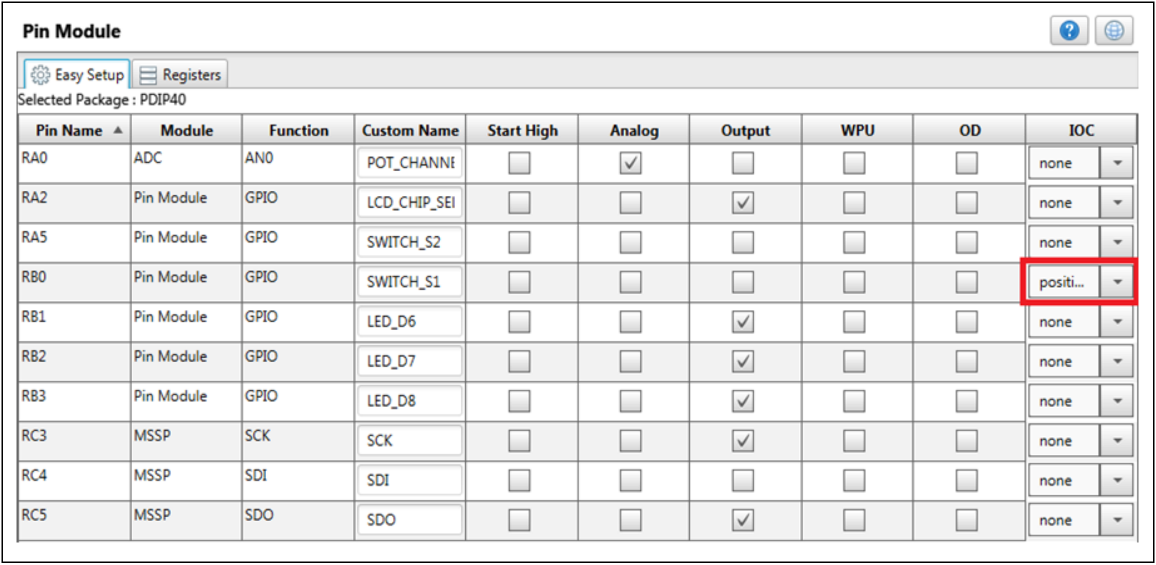
```
INTERRUPT_TMR0InterruptEnable();
```

Equivalent:

```
TMR0IE = 1;
```

This line enables interrupts to occur every time Timer0 overflows.

FIGURE 9-2: COMPOSER AREA – GPIO MODULE



Selected Package : PDIP40

Pin Name	Module	Function	Custom Name	Start High	Analog	Output	WPU	OD	IOC
RA0	ADC	AN0	POT_CHANNE	☐	✓	☐	☐	☐	none
RA2	Pin Module	GPIO	LCD_CHIP_SEI	☐	☐	✓	☐	☐	none
RA5	Pin Module	GPIO	SWITCH_S2	☐	☐	☐	☐	☐	none
RB0	Pin Module	GPIO	SWITCH_S1	☐	☐	☐	☐	☐	positi...
RB1	Pin Module	GPIO	LED_D6	☐	☐	✓	☐	☐	none
RB2	Pin Module	GPIO	LED_D7	☐	☐	✓	☐	☐	none
RB3	Pin Module	GPIO	LED_D8	☐	☐	✓	☐	☐	none
RC3	MSSP	SCK	SCK	☐	☐	✓	☐	☐	none
RC4	MSSP	SDI	SDI	☐	☐	☐	☐	☐	none
RC5	MSSP	SDO	SDO	☐	☐	✓	☐	☐	none

To enable IOC on positive edge, ‘positive’ should be selected from the port’s IOC dropdown menu.

Interrupt Service Routine

If any interrupts occur, the program will jump to this subroutine and identify which interrupt occurred by checking which flag is set and if the corresponding enable bit is set. If both conditions are met, it will proceed to the function designated to handle the interrupt.

```
#include "interrupt_manager.h"
#include "mcc.h"

void interrupt INTERRUPT_InterruptManager(void) {
    // interrupt handler
    if(INTCONbits.TMR0IE == 1 && INTCONbits.TMR0IF == 1)
    {
        TMR0_ISR();
    }
    else if(INTCONbits.IOCIE == 1 && INTCONbits.IOCIF == 1)
    {
        PIN_MANAGER_IOC();
    }
    else
    {
        //Unhandled Interrupt
    }
}
```

IOC Interrupt Handler (PIN_MANAGER_IOC)

```
void PIN_MANAGER_IOC(void)
{
    // interrupt on change for pin IOCBF0
    if(IOCBFbits.IOCBF0 == 1)
    {
        IOCBF0_ISR();
    }
}
void IOCBF0_ISR(void) {

    // Call the interrupt handler for the callback registered at
    // runtime
    if(IOCBF0_InterruptHandler)
    {
        IOCBF0_InterruptHandler();
    }
    IOCBFbits.IOCBF0 = 0;
}
```

The `'Lab_IOC_ISR()'` function inside the `'IOCBF0_SetInterruptHandler()'` will be called.

IOC Interrupt Handler (Lab_IOC_ISR)

```
void Lab_IOC_ISR(void){

    // Toggle the direction of the rotation
    rotateDirection ^= 1;

    // If the rotation is to the right, it will display right
    if(rotateDirection == RIGHT){
        LCD_GoTo(1,0);
        LCD_WriteString("Direction: Right");
    }
    // If the rotation is to the right, it will display left
    else if(rotateDirection == LEFT){
        LCD_GoTo(1,0);
        LCD_WriteString("Direction: Left ");
    }
}
```

Every time the rotation changes its direction, the current direction is displayed on the LCD.

Timer0 Interrupt Service Routine (TMR0_ISR)

When using MCC to set up interrupts, the ISR handler function is generated with the source file of the peripheral (i.e. Timer0 ISR function is found in `tmr0.c`). You might need to modify the MCC-generated file to include your custom code to handle the interrupt and to make sure that all necessary headers are included for your code to work. The following code is a custom code that rotates the LED to the right every time the timer rolls over.

```
void TMR0_ISR(void) {
    static volatile uint16_t CountCallBack = 0;

    // clear the TMR0 interrupt flag
    INTCONbits.TMR0IF = 0;

    TMR0 = timer0ReloadVal;

    // callback function - called every 20th pass
    if (++CountCallBack >= TMR0_INTERRUPT_TICKER_FACTOR) {
        // ticker function call
        TMR0_CallBack();

        // reset ticker counter
        CountCallBack = 0;
    }

    // add your TMR0 interrupt custom code
}
```

Timer0 CallBack function (TMR0_Callback)

If the maximum period of Timer0 is not enough, MCC’s `TMR0_CallBack()` function can be used to execute at the user-provided interrupt handling code on every nth Timer0 overflow (see Figure 9-2).

```
void TMR0_CallBack(void) {

    if(TMR0_InterruptHandler)
    {
        TMR0_InterruptHandler();
    }
}
```

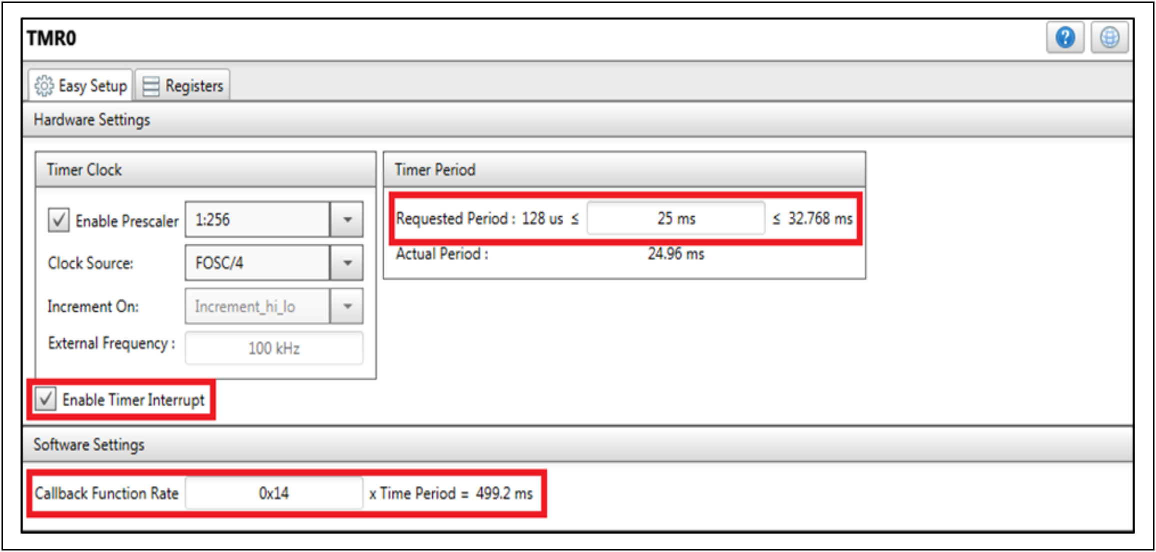
The `‘Lab_Timer0_ISR()’` function inside the `‘TMR0_SetInterruptHandler()’` will be called.

Timer0 Interrupt Handler (Lab_Timer0_ISR)

```
void Lab_Timer0_ISR(void){

    // if the direction of the LEDs rotation is to the right
    if(rotateDirection == RIGHT){
        // Rotate the LEDs to the right
        LEDs_ShiftRight();
        // Increment the rotateCounter
        rotateCounter++;
        // If the last LED (D8) is lit, go back to the first
        // LED (D6)
        if(rotateCounter > 8){
            LED_D6_SetHigh();
            rotateCounter = 6;
        }
    }

    // if the direction of the LEDs rotation is to the left
    else if(rotateDirection == LEFT){

        // Rotate the LEDs to the left
        LEDs_ShiftLeft();
        // Decrement the rotateCounter
        rotateCounter--;
        // If the last LED (D6) is lit, go back to the first
        // LED (D8)
        if(rotateCounter < 6){
            LED_D8_SetHigh();
            rotateCounter = 8;
        }
    }
}
```

The `'Lab_Timer0_ISR()'` function allows the shifting of the set bit along PORTB to occur every ~500ms, either to the left or right.

FIGURE 9-3: COMPOSER AREA – TIMER0 MODULE


TMR0

Easy Setup | Registers

Hardware Settings

Timer Clock

- [✓] Enable Prescaler: 1:256
- Clock Source: FOSC/4
- Increment On: Increment_hi_lo
- External Frequency: 100 kHz

[✓] Enable Timer Interrupt

Timer Period

- Requested Period : 128 us ≤ 25 ms ≤ 32.768 ms
- Actual Period : 24.96 ms

Software Settings

Callback Function Rate 0x14 x Time Period = 499.2 ms

As seen in Figure 9-2, Timer0 overflows every ~25ms. But since 500ms is needed for the rotation, a callback function rate of 20 is configured.

For PIC18F devices:

Main Program (Interrupt Set-up)

The initialization of interrupts may vary for PIC18 devices. For PIC18, there is no `IOCIE` bit in the interrupt SFRs, hence Interrupts-on-Change are configured differently.

```
void Interrupt(void)
{
    INTERRUPT_GlobalInterruptEnable();
    INTERRUPT_PeripheralInterruptEnable();
    // Setup the interrupt on change for S1
    INTERRUPT_InterruptOnChangeEnable();
    // Enable Interrupts on the Positive Edge
    INTERRUPT_IOCPositiveEnable();
    // Enable the TMR0 Interrupts
    INTERRUPT_TMR0InterruptEnable();

    //Code other initializations and wait for an interrupt to occur
    while(1) {}
}
```

```
INTERRUPT_InterruptOnChangeEnable();

Equivalent:
INTCONbits.INT0IE = 1;
```

Interrupts-on-Change (IOC) are available for a few selected Rxx pins (refer to the PIC18 device datasheet for these pins). These are enabled by setting the respective `INTxIE` bit.

```
INTERRUPT_IOCPositiveEnable();

Equivalent:
INTCON2bits.INTEDG0 = 1;
```

Either positive or negative edge can be used by setting or clearing the `INTEDGx` bit respectively.

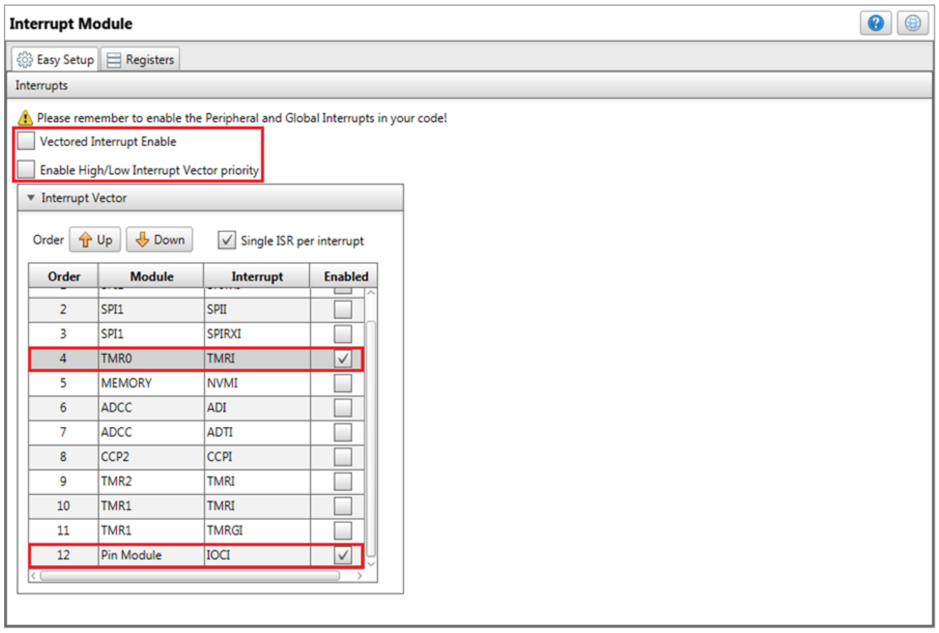
Interrupt Service Routine

```
// If the Interrupt on Positive Edge is enabled:
if(INTCON2bits.INTEDG0 == 1 && INTCONbits.INT0F == 1) {
    // Change the direction of rotation
    direction ^= 1;

    if(direction == RIGHT){
        // Display the direction of rotation to the LCD
        LCDGoto(1,0);
        WriteLCDString((char*)"Direction: Right");
    }else{
        if(direction == LEFT){
            // Display the direction of rotation to the LCD
            LCDGoto(1,0);
            WriteLCDString((char*)"Direction: Left ");}
    }
    // Clear the interrupt on change flag
    INTCONbits.INT0F = 0;
}
```

The only difference between PIC16 and PIC18 in this section is the use of `INTEDG0` and `INT0F` for PIC18’s IOC. These can be inserted into `pin_manager.c` by replicating PIC16’s IOC Interrupt Handler, or into `interrupt_manager.c` under Unhandled Interrupts.

Shown below are the steps in configuring the Interrupt module (FIGURE 9-4) and Timer0 module (FIGURE 9-5) through the use of MCC.

FIGURE 9-4: COMPOSER AREA – INTERRUPT MODULE



Interrupt Module

Easy Setup | Registers

Interrupts

Please remember to enable the Peripheral and Global Interrupts in your code!

☐ Vectored Interrupt Enable

☐ Enable High/Low Interrupt Vector priority

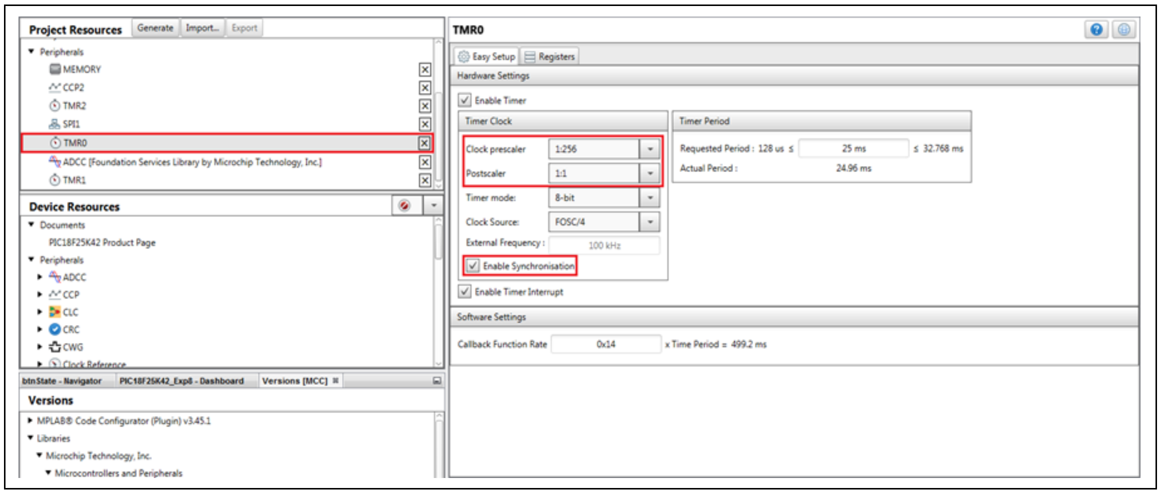
Interrupt Vector

Order Up Down ✓ Single ISR per interrupt

Order	Module	Interrupt	Enabled
2	SPI1	SPII	☐
3	SPI1	SPIRXI	☐
4	TMR0	TMRI	✓
5	MEMORY	NVMI	☐
6	ADCC	ADI	☐
7	ADCC	ADTI	☐
8	CCP2	CCPI	☐
9	TMR2	TMRI	☐
10	TMR1	TMRI	☐
11	TMR1	TMRGI	☐
12	Pin Module	IOCI	✓

For PIC18, it has an Interrupt Priority feature (refer to the PIC18 device datasheet) that allows most interrupt sources to be assigned a high or low priority level.

FIGURE 9-5: MCC WINDOW – TIMER0 MODULE



[✓] Enable Timer

Clock prescaler	1:256
Postscaler	1:1
Timer mode:	8-bit
Clock Source:	FOSC/4
External Frequency :	100 kHz

[✓] Enable Synchronisation

Requested Period : 128 us ≤ 25 ms ≤ 32.768 ms

Actual Period : 24.96 ms

[✓] Enable Timer Interrupt

Callback Function Rate 0x14 x Time Period = 499.2 ms

For PIC18, Timer0 can operate as either an 8 bit or a 16 bit timer/counter as seen in Figure 9-4,

Note: For some devices in the PIC18fxxK40 family, the `'Enable Synchronization'` option must be disabled for proper interrupt operation.

LESSON 10: SLEEP/WAKEUP

10.1 Introduction

This lesson will introduce the Sleep mode. The `SLEEP()` function is used to put the device into a low-power standby mode.

10.2 Hardware Effects

Upon entering this lab, a countdown timer will be displayed on the LCD and the device will be put into Sleep after 5 seconds. The device will then wake up after 8 seconds and the LCD will display “I'm now awake!” on the second line.

10.3 Summary

Sleep is one of the power-saving modes in PIC microcontrollers. In Sleep mode, the main CPU clock and most peripheral clock sources are shut down, bringing the device to a low power state. The current device state is maintained, including RAM, SFRs and the Program Counter (PC). Current consumption during Sleep typically range from 50-100 nA and varies depending upon the device used. There are still peripherals that can run during Sleep such as ADC, Timer and comparator modules. Refer to the specific device data sheet for a complete list of peripherals and the effect of Sleep on each of them.

Sleep wake-up events include Reset, interrupts, or a Watchdog Timer (WDT) time out. If any of these events occur during Sleep, the device will resume to normal operation and will execute the next instruction from where it left before entering Sleep.

For this lesson, a WDT is used as the wake-up event. The WDT times out every 8 seconds.

10.4 New Registers

Register	Purpose
WDTCON	Contains enable and period select bits for the watchdog timer

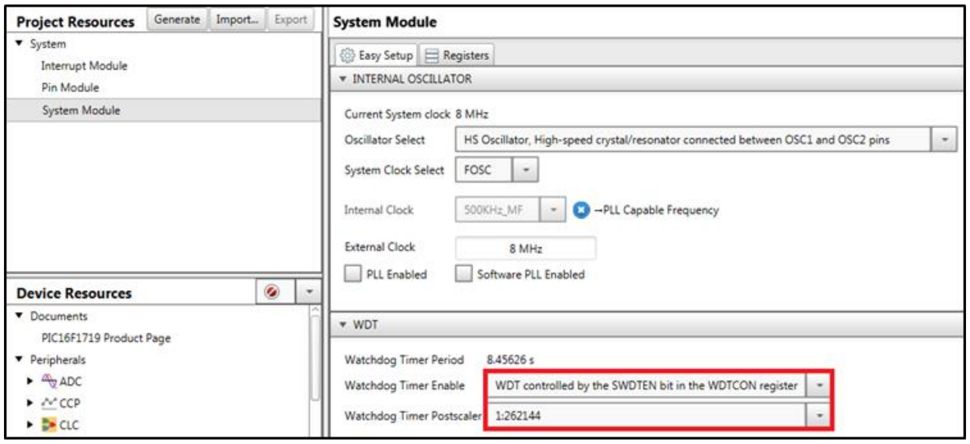
WDTCON

This register contains the software enable (SWDTEN) bits and the period select bits for the watchdog timer. Note that for some devices, the period select bits are found in the CONFIG register. The period select bits determine how long a program runs before the Watchdog timer resets the microcontroller. Values range to produce periods from 1 ms to 256 s and varies depending on the device used.

10.5 MCC Setup

The Watchdog Timer settings can be configured in the System Module. This lab uses the configuration shown in Figure 10-1. The option selected in ‘Watchdog Timer Enable’ allows the programmer to enable or disable the watchdog timer via software. The option selected in ‘Watchdog Timer Postscaler’ causes the watchdog timer to timeout every 8 seconds (this is only an approximation; the actual value is shown in ‘Watchdog Timer Period’). For some devices, the WDT clock reference is scaled by 128 before entering the postscaler. In this case, the postscaler setting in MCC should be set to 1:2048 for a WDT reset to occur every 8 seconds. See the specific device datasheet for more information.

FIGURE 10-1: MCC WINDOW – SYSTEM CONFIGURATIONS FOR WDT



Current System clock 8 MHz
Oscillator Select: HS Oscillator, High-speed crystal/resonator connected between OSC1 and OSC2 pins
System Clock Select: FOSC
Internal Clock: 500KHz_MF →PLL Capable Frequency
External Clock: 8 MHz
PLL Enabled
Software PLL Enabled
Watchdog Timer Period: 8.45626 s
Watchdog Timer Enable: WDT controlled by the SWDTEN bit in the WDTCON register
Watchdog Timer Postscaler: 1:262144

10.6 MCC Instructions

Instruction	Purpose
`SLEEP()`	Puts the device in a low power mode.

10.7 C Language

A sample code written in C for the Sleep/Wakeup lab is provided below.

EXAMPLE 10.1: C CODE FOR “SLEEP/WAKEUP” LAB

```
/**
  Section: Included Files
 */
#include "../../mcc_generated_files/pin_manager.h"
#include "../../labs.h"
#include "../../lcd.h"

/**
  Section: Macro Declaration
 */
#define COUNTDOWN_MAX     5   // seconds
#define WDT_Enable()     (WDTCONbits.SWDTEN = 1)
#define WDT_Disable()    (WDTCONbits.SWDTEN = 0)

/*
                             Application
 */
void SleepWakeUp(void){
    LEDs_SetLow();
    LED_D7_SetHigh();
    LCD_GoTo(0,0);
    LCD_WriteString(" Sleep/Wake Up  ");

    WDT_Enable();

    uint8_t wdtTimer = COUNTDOWN_MAX;
    for(uint8_t i = 0; i < COUNTDOWN_MAX; i++){
        LCD_GoTo(1,0);
        LCD_WriteString(" Sleeping in    ");
        LCD_GoTo(1,13);
        LCD_WriteByte((wdtTimer)%10+'0');

        wdtTimer--;
        __delay_ms(1000);
    }

    LCD_GoTo(1,0);
    LCD_WriteString("Wait for 8 secs ");

    SLEEP();

    while (1) {
        // Wait for 8s for the WDT time-out; and the LEDs will toggle
        LED_D7_LAT = LOW;
        LED_D6_LAT = LED_D8_LAT = HIGH;

        LCD_GoTo(1,0);
        LCD_WriteString(" I'm now awake! ");

        WDT_Disable();
    }
}
```

```
WDT_Enable();
```

Equivalent:

```
WDTCONbits.SWDTEN = 1;
```

Setting the SWDTEN bit enables the WDT to run for a period of time defined by the WDT Period Select bits configured in MCC.

```
uint8_t wdtTimer = COUNTDOWN_MAX;
for(uint8_t i = 0; i < COUNTDOWN_MAX; i++){
    LCD_GoTo(1,0);
    LCD_WriteString(" Sleeping in    ");
    LCD_GoTo(1,13);
    LCD_WriteByte((wdtTimer)%10+'0');

    wdtTimer--;
    __delay_ms(1000);
}

LCD_GoTo(1,0);
LCD_WriteString("Wait for 8 secs ");
```

These set of codes allows the device to countdown for 5 seconds before entering Sleep mode. Afterwards, a string is sent to the LCD to instruct the user to wait for 8 seconds before the device wakes up from sleep.

```
SLEEP();
```

This function puts the device into Sleep mode. The WDT will be cleared as the device enters Sleep and will continue to run while sleeping. The Oscillator Start-up Timer will be enabled after returning from Sleep.

Note: For some devices, the Fail-Safe Clock Monitor will be active during oscillator start-up and the system clock will be switched to an internal clock. The system clock can be switched back to the external crystal oscillator by writing to the oscillator control register. See the Fail-Safe Clock Monitor section of the device datasheet for more information.

```
// Wait for 8s for the WDT time-out; and the LEDs will toggle
LED_D7_LAT = LOW;
LED_D6_LAT = LED_D8_LAT = HIGH;

LCD_GoTo(1,0);
LCD_WriteString(" I'm now awake! ");
```

The user will then be notified that a wake-up event has occurred when the LEDs toggle and the LCD displays that the device is already awake.

LESSON 11: EEPROM

11.1 Introduction

EEPROM is nonvolatile memory, meaning that it does not lose its contents when power is cut off. This is unlike RAM, which will lose its value when no power is applied. The EEPROM is useful for storing data that must still be present after a power interruption. It is also convenient to use if the entire RAM space is used up. Writes and reads to the EEPROM are relatively quick, and are much faster than program memory operations.

11.2 Hardware Effects

Press the switch to show the ADC reading on the LCD and save it to the on-chip Data EEPROM. The LCD displays “EEPROM” on the first line and “Value = *x*” on the second line, where *x* corresponds to the saved ADC result in the EEPROM.

11.3 Summary

When the lesson is first programmed, no LEDs will light up and the value on the LCD will be the value of registers for the previous lab. This will not change yet, even with movement of the potentiometer (R25). When the switch is pressed, the 3 MSBs of the ADC reading at that instant will be displayed on the LEDs, and the LCD will display the decimal equivalent. Each press of the switch saves the ADC value into EEPROM.

11.4 New Registers

Register	Purpose
EECON1 and EECON2	Controls EEPROM read/write access
EEDATH:EEDATL	Data register pair
EEADRH:EEADRL	Address register pair

EECON1 and EECON2
EECON1 contains specific bits used to access and enable EEPROM. Commonly used bits are EEPGD to determine if the PIC® will access EEPROM or flash memory; RD and WR bits to initiate read and write respectively; and WREN bit to enable write operation. EECON2 contains the Data EEPROM Unlock Pattern bits. A specific pattern must be written to this register for unlocking writes.

EEDATH:EEDATL
EEDATH:EEDATL form a register pair which holds the 14-bit data for read/write.

EEADRH:EEADRL
EEADRH:EEADRL form a register pair which holds the 15-bit address of the program memory location being read.

Note that for some latest devices, the above registers have been replaced by the following:

Register	Purpose
NVMCON1 and NVMCON2	Controls EEPROM read/write access
NVMDATH:NVMDATL	Data register pair
NVMADRH:NVMADRL	Address register pair

NVMCON1 and NVMCON2

NVMCON1 contains specific bits used to access and enable nonvolatile memory, including EEPROM. Commonly used bits are NVMREG to determine if the PIC® will access program flash memory, EEPROM, or User/Device IDs and Configuration bits; RD and WR bits to initiate read and write respectively; and WREN bit to enable write operation. NVMCON2 contains the NVM Unlock Pattern bits. A specific pattern must be written to this register for unlocking writes.

NVMDATH:NVMDATL

NVMDATH:NVMDATL form a register pair which holds the NVM data for read/write. Some devices only have a single register called NVMDAT for the same purpose.

NVMADRH:NVMADRL

NVMADRH:NVMADRL form a register pair which holds the address of the NVM location being read. Note that some devices do not implement the NVMADRH register; refer to the device datasheet for more information.

11.5 MCC Setup

Add ‘MEMORY’ to the Project Resources window.

11.6 MCC Instructions

Instruction	Purpose
`DATAEE_WriteByte(uint8_t bAdd, uint8_t bData)`	Writes the value stored in `bData` to EEPROM data memory address `bAdd`
`DATAEE_ReadByte(uint8_t bAdd)`	Reads from EEPROM address `bAdd`

11.7 C Language

A sample code written in C for EEPROM is provided below.

EXAMPLE 11.1: C CODE FOR “EEPROM” LAB

```
/**
  Section: Included Files
 */
#include "../../mcc_generated_files/memory.h"
#include "../../mcc_generated_files/pin_manager.h"
#include "../../mcc_generated_files/adc.h"
#include "../../mcc_generated_files/interrupt_manager.h"
#include "../../labs.h"
#include "../../lcd.h"
```

```
/**
  Section: Local Function Prototypes
 */
void Lab_EEPROM_ISR(void);


/**
  Section: Macro Declaration
 */
#define EEPROM_ADDR    0x80      // EEPROM starting address


/**
  Section: Local Variable Declarations
 */
uint16_t adcResult;


/*
                             Application
 */
void EEPROM(void) {
    LEDs_SetLow();

    INTERRUPT_GlobalInterruptEnable();
    INTERRUPT_PeripheralInterruptEnable();
    INTERRUPT_InterruptOnChangeEnable();
    INTERRUPT_IOCPositiveEnable();

    IOCBF0_SetInterruptHandler(Lab_EEPROM_ISR);

    LCD_GoTo(0, 0);
    LCD_WriteString("     EEPROM        ");

    adcResult = 0;

    while (1) {
        uint8_t readData;

        // Display the stored data from the EEPROM onto the LEDs
        // MSB = LED_D8, LSB = LED_D6
        readData = DATAEE_ReadByte(EEPROM_ADDR);
        LEDs = readData << 1;

        // Display the stored data from the EEPROM onto the LCD
        LCD_GoTo(1, 0);
        LCD_WriteString("   Value =      ");
        LCD_GoTo(1, 11);
        LCD_WriteByte((readData) % 10 + '0');
    }
}
```

```
INTERRUPT_GlobalInterruptEnable();
INTERRUPT_PeripheralInterruptEnable();
INTERRUPT_InterruptOnChangeEnable();
INTERRUPT_IOCPositiveEnable();
```

These macros enable the interrupts used for this lab. These should be disabled before moving to another lab.

```
IOCBF0_SetInterruptHandler(Lab_EEPROM_ISR);
```

This changes the interrupt handler used in the Interrupt Lab, `Lab_IOC_ISR()`, to `Lab_EEPROM_ISR()`. On an IOC event, the function `Lab_EEPROM_ISR()` will be called instead of `Lab_IOC_ISR()`.

```
readData = DATAEE_ReadByte(EEPROM_ADDR);
```

Upon entering a loop which waits for the IOC, the device will continuously read the EEPROM at a single and unchanging address and display this on both the LEDs and LCD.

IOC Interrupt Handler (Lab_EEPROM_ISR)

```
void Lab_EEPROM_ISR(void) {
    // Debounce by delaying
    __delay_ms(20);

    // Get the top 3 MSBs and display it in the LEDs
    adcResult = ADC_GetConversion(POT_CHANNEL) >> 7;
    LEDs = adcResult << 1;

    // Save the value to EEPROM and wait for the next IOC event
    DATAEE_WriteByte(EEPROM_ADDR, adcResult);
}
```

```
adcResult = ADC_GetConversion(POT_CHANNEL) >> 7;
DATAEE_WriteByte(EEPROM_ADDR, adcResult);
```

Upon entering the ISR with IOC, the device takes the ADC reading from the potentiometer and stores its 3 MSBs into the EEPROM on address `EEPROM_ADDR` using `DATAEE_WriteByte()`.

LESSON 12: HIGH-ENDURANCE FLASH (HEF)

12.1 Introduction

In this lesson, we will discuss High-Endurance Flash (HEF) Memory, an alternative to Data EEPROM memory present in many devices. As we progress, we will also discuss the similarities and differences between these two as well as the purpose and set-up procedures to use the available HEF memory block on devices.

12.2 Hardware Effects

Press the switch to show the voltage reading on the LCD and save it to the on-chip High-Endurance Flash memory. The LCD displays “HEF” on the first line and “Voltage = *x.xx* V” on the second line, where *x.xx* corresponds to the actual voltage reading.

12.3 Summary

High-Endurance Flash (HEF) Memory is a type of non-volatile memory much like the Data EEPROM. Data stored in this type of memory is retained in spite of power outages. HEF’s advantage over regular Flash Memory lies in its superior Erase-Write cycle endurance. While regular Flash could only sustain around 10,000 E/W cycles before breaking down, HEF can go for around 100,000 E/W cycles, within the range of average EEPROM endurance. Between true EEPROM and HEF, the difference lies in how operations are handled in both types of memory. In HEF, erase and write operations are performed in fixed blocks as opposed to data EEPROMs that are designed to allow byte-by-byte erase and write. Another difference is that writing to HEF stalls the processor for a few milliseconds as the MCU is unable to fetch new instructions from the Flash memory array. This is in contrast to true data EEPROMs which do not stall MCU executions during a write cycle.

12.4 New Registers

There are no new registers for this lab. Refer to the previous labs for the registers used.

12.5 MCC Setup

Add ‘MEMORY’ to the Project Resources window.

12.6 MCC Instructions

Instruction	Purpose
`FLASH_ReadWord(uint16_t flashAddr)`	Reads a word from flash memory.
`FLASH_WriteWord(uint16_t flashAddr, uint16_t *ramBuf, uint16_t word)`	Writes a word to flash memory.

12.7 C Language

A sample code for High Endurance Flash for PIC16F devices is provided below.

EXAMPLE 12.1: C CODE FOR “HEF” LAB

```
/**
  Section: Included Files
 */
#include "../../mcc_generated_files/memory.h"
#include "../../mcc_generated_files/pin_manager.h"
#include "../../mcc_generated_files/adc.h"
#include "../../labs.h"
#include "../../lcd.h"

/**
  Section: Macro Declaration
 */
#define ADDR    0x3F80    // HEF address

/**
  Section: Local Variable Declarations
 */
uint16_t adcResult;

/*
                             Application
 */
void HEF(void) {
    LEDs_SetLow();

    LCD_GoTo(0, 0);
    LCD_WriteString("      HEF       ");
    LCD_GoTo(1, 0);
    LCD_WriteString("Voltage = 0.00 V");

    while (1) {
        uint16_t readData;
        uint16_t voltVal;
        uint16_t HefAddr = ADDR;
        uint16_t Buf[ERASE_FLASH_BLOCKSIZE];

        if (SWITCH_S1_PORT == LOW) {
            // Debounce by delaying
            __delay_ms(120);

            // Check if S1 is still pressed
            if (SWITCH_S1_PORT == LOW) {

                adcResult = ADC_GetConversion(POT_CHANNEL);
                adcResult /= 2;

                FLASH_WriteWord(HefAddr, Buf, adcResult);

                readData = FLASH_ReadWord(HefAddr);
                voltVal = readData;

                // Display the voltage value to the LCD
                LCD_GoTo(1, 10);
                LCD_WriteByte((voltVal / 100) + '0');
                LCD_WriteByte('.');
                LCD_WriteByte((voltVal / 10) % 10 + '0');
                LCD_WriteByte((voltVal % 10) + '0');
            }
        }
    }
}
```

```
adcResult = ADC_GetConversion(POT_CHANNEL);

adcResult /= 2;
```

The raw ADC result is divided by 2 to get the equivalent voltage value from the potentiometer.

```
FLASH_WriteWord(HefAddr, Buf, adcResult);
```

This is the function that writes the contents of `adcResult` to HEF memory address `HefAddr`. `Buf` is a pointer to an array containing at least ERASE_BLOCK_SIZE elements. This array is used to store the current contents of a block (also called *row*) of HEF memory prior to performing the write operation; see the datasheet for more information.

Note: XC8 treats HEF memory just like regular program memory and uses both to store executable code. To reserve an address range for user data, use the option

`--ROM=default,-`*`range`*

Or in MPLAB X, go to the XC8 linker option in the Project Properties window and select ‘Memory model’. In the ‘ROM ranges’ field, enter `default,-f80-fff` to reserve the specified address range.

```
readData = FLASH_ReadWord(HefAddr);
```

This function returns the contents of HEF memory address `HefAddr` to `readData`. The value is then used for subsequent LCD output.

NOTES: