Introduction

The following labs are designed for Microchip’s Curiosity board. The Curiosity Development Board supports Microchip's 8, 14 and 20-pin 8-bit PIC® MCUs. The MPLAB X project that you downloaded from the Curiosity Website contains 10 lab exercises that demonstrate the myriad basic capabilities of PIC® devices and can also be used to test the condition of your board. You can progress through each of the labs by simply pressing the S1 button of your board. In this document, you can find tutorials on the basic tasks and peripherals of the PIC® devices. You can also find information on the different registers and bits associated with the peripherals that might be of use in your next microcontroller project. This document also details steps on how to set-up and generate your code using the MPLAB Code Configurator which was also used in the creation of these labs. Also provided in this document are the basic code snippets upon which the Curiosity lab codes were built upon and a discussion of said codes. The rest of the document is a description of what each lab does and what the user should see on the LEDs. All the labs are written in C language compatible with the latest XC8 compilers.

LESSON 1: HELLO WORLD (TURN ON AN LED)

Introduction

The first lesson shows how to turn on an LED.

Hardware Effects

D4 will light up and stay lit.

Summary

The LEDs are connected to the input-output (I/O) pins. First, the I/O pin must be configured to be an output. In this case, when one of these pins is driven high (D4 = 1), the LED will turn on. These two logic levels are derived from the power pins of the PIC® MCU. Since the PIC®'s power pin (VDD) is connected to 5V and the source (VSS) to ground (0V), a logic level of ‘1’ is equivalent to 5V, and a logic level of ‘0’ is 0V.

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.

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
`D4_SetHigh()`	Make the bit value of D4 (LATA5) a ‘1’ (5V)
`D4_SetLow()`	Make the bit value of D4 (LATA5) a ‘0’ (0V)

EXAMPLE 1.1: SETTING A BIT INTO ‘1’

```
D4_SetHigh();
```

Before Instruction:
`LATA5 = 0;`
After Instruction:
`LATA5 = 1;`

EXAMPLE 1.2: SETTING A BIT INTO ‘0’

```
D4_SetLow();
```

Before Instruction:
`LATA5 = 1;`
After Instruction:
`LATA5 = 0;`

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

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

void HelloWorld(void)
{
   //Turn ON D4
```

```
    D4_SetHigh();
}
```

```
//
```

This starts a comment. Any of the following text on this line is ignored by the compiler. Be sure to place lots of these in your code for readability.

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

This header file `mcc.h` is generated automatically by the MPLAB Code Configurator (MCC). It provides implementations for driver APIs for all modules selected in the MCC GUI.

```
#include "../../global.h"
```

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

```
    D4_SetHigh();
```

This statement calls the function `D4_SetHigh()`. `D4_SetHigh()` turns on LED D4.

```
D4_SetHigh();
```

Equivalent:

```
#define D4_SetHigh()    do { LATA5 = 1; } while(0)
```

This function is defined in `pin_manager.h` under the *MCC Generated Files* folder. It sets the LAT register of RA5 to 1.

.

LESSON 2: BLINK

Introduction

This lesson blinks the same LED used in the previous lesson (D4).

Hardware Effects

D4 blinks at a rate of approximately 1.5 seconds.

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 code and compiles it into assembly language 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 code to be executed. 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 7: TIMER1**.**

New Registers

This utilizes Timer1 registers which will be discussed in Lesson 7: TIMER1.

MCC Instructions

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

Instruction	Purpose
`D4_Toggle()`	Changes the bit value of D4 (LATA5) from ‘`0`’ to ‘`1`’, or ‘`1`’ to ‘`0`’

EXAMPLE 2.1: TOGGLING A BIT

```
D4_Toggle();
```
Before Instruction:
```
LATA5 = 0;
```
After Instruction:
```
LATA5 = 1;
```

Or

Before Instruction:
```
LATA5 = 1;
```
After Instruction:
```
LATA5 = 0;
```

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/mcc.h"
#include "../../global.h"

/*
                             Application
 */
void Blink(void)
{
    //Wait for Timer1 to overflow
    while(!TMR1IF);

    //Reload the initial value of TMR1
    TMR1_Reload();

    //Increment the counter variable
    counter++;

    //Wait for third overflow for 1.5 secs delay
    if(counter == 3){
        //Toggle D4
        D4_Toggle();

        //Reset counter to 0
        counter = 0;
    }

    //Clear TMR1 Overflow flag
    TMR1IF = 0;
}
```

```
    //Wait for Timer1 to overflow
    while(!TMR1IF);

    //Reload the initial value of TMR1
    TMR1_Reload();
```

The program will first wait for the Timer1 flag (for approximately 500 ms) to be set before executing the next instructions, and will reload the same value of 500 ms to the Timer1 (see **LESSON 7**).

```
//Increment the counter variable
counter++;

//Wait for third overflow for 1.5 secs delay
if(counter == 3){
    //Toggle D4
    D4_Toggle();

    //Reset counter to 0
    counter = 0;
}
```

A global variable ‘`counter`’ increments every time `TMR1IF` is set until it reaches a value of ‘`3`’. This signifies that Timer1 has overflowed after 500 ms three times for a total of 1.5 seconds, before D4 is toggled. ‘`counter`’ is then reset to ‘`0`’ and the process is repeated.

```
D4_Toggle();
```

Equivalent:

```
#define D4_Toggle()   do { LATA5 = ~LATA5; } 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 RA5 PORT data latch (LATA5), making the pin “high” if previously “low” or vice versa.

LESSON 3: ROTATE (MOVING THE LIGHT ACROSS LEDS)

Introduction

This lesson would build on Lessons 1 and 2, which showed how to light up a LED and then make it blink with using loops. This lesson incorporates four onboard LEDs (D4, D5, D6 and D7) and the program will light each LED up in turn.

Hardware Effects

Program will each light up D4, D5, D6 and D7 in turn every 500 milliseconds. Once D7 is lit, D4 lights up and the pattern repeats.

Summary

In C, we use Binary Left Shift and Right Shift Operators (<< and >>, respectively) to move bits around in the registers. The shift operations are 9-bit operations involving the 8-bit register being manipulated and the Carry bit in the STATUS register as the ninth bit. With the rotate instructions, the register contents are rotated through the Carry bit.

For example, for a certain register `rotateReg`, if we want to push a ‘`1`’ into the LSB of the register and have the rest of the bits shift to the left, we can use the Binary Left Shift Operator (`<<`). We would first have to set up the Carry bit with the value that we want to push into the register before we execute shift instruction, as seen in Figure 3-1A. The result of the operation is seen in Figure 3-1B.

FIGURE 3-1: LEFT SHIFT BINARY OPERATION

rotateReg before instruction:

<7:0>							
0	1	0	0	1	1	0	0

STATUS register before instruction:

IRP	RP1	RP0	!TO	!PD	Z	DC	C
0	1	0	0	1	1	0	1

(A)

rotateReg after instruction:

<7:0>							
1	0	0	1	1	0	0	1

STATUS register after instruction:

IRP	RP1	RP0	!TO	!PD	Z	DC	C
0	1	0	0	1	1	0	0

(B)

Similarly, if we want to push a ‘`1`’ into the MSB of the register and have the rest of the bits shift to the right, we can use the Binary Right Shift Operator (`>>`). We would first have to set up the Carry bit with the value that we want to push into the register before we execute shift instruction, as seen in Figure 3.2A. The result of the operation is seen in Figure 3.2B.

FIGURE 3-2: RIGHT SHIFT BINARY OPERATION

rotateReg before instruction:

<7:0>							
0	1	0	0	1	1	0	0

STATUS register before instruction:

IRP	RP1	RP0	!TO	!PD	Z	DC	C
0	1	0	0	1	1	0	1

(A)

rotateReg after instruction:

<7:0>							
1	0	1	0	0	1	1	0

STATUS register after instruction:

IRP	RP1	RP0	!TO	!PD	Z	DC	C
0	1	0	0	1	1	0	0

(B)

New Register

Register	Purpose
STATUS	Multi-purpose; depends on which bits are accessed.

STATUS
The STATUS register contains the arithmetic status of the ALU (Arithmetic Logic Unit), the Reset status and the bank select bits for data memory. For more details, please see the device datasheet.

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/mcc.h"
#include "../../global.h"

/*
                              Application
 */
void Rotate(void)
{
    // Begin with D4 High
    D4_LAT = 1;

    // Initialize temporary register to begin at 1
    rotateReg = 1;

    while(1){
        // Call the Delay() function and keep LED on for 0.5 secs
        Delay();

        // Shift value in temp register to the right by 1 bit
            rotateReg <<= 1;

        // If the last LED has been lit, restart the pattern
        if(rotateReg == 16)
            rotateReg = 1;

        // Determine which LED will light up
        // ie. which bit in the register the 1 has rotated to.
        D4_LAT = rotateReg & 1;
        D5_LAT = (rotateReg & 2) >> 1;
        D6_LAT = (rotateReg & 4) >> 2;
        D7_LAT = (rotateReg & 8) >> 3;
    }
}

// Delay function to keep the LED on for 0.5 secs before rotating
void Delay(void)
{
    int i = 0;
    for(i=0;i<25;i++){
        __delay_ms(20);
    }
}
/**
 End of File
 */
```

```
D4_LAT = rotateReg & 1;
D5_LAT = (rotateReg & 2) >> 1;
D6_LAT = (rotateReg & 4) >> 2;
D7_LAT = (rotateReg & 8) >> 3;
```

The above statements are used to reflect the value stored in `rotateReg` onto the LEDs. The Bitwise AND operator is used to determine whether the LEDs output is high or low. Then the bits are shifted with respect to its position. The following shows the bitwise AND operation on how the value of `rotateReg` (0b1000 in this example) is reflected to the LEDs.

```
D4_LAT = rotateReg & 1;

rotateReg:   1000
1        : & 0001
----------------------
D4_LAT   :   0000 (OFF)


D5_LAT = (rotateReg & 2) >> 1;

rotateReg:   1000
2        : & 0010
---------------------
             0000

>> 1     :   0000

----------------------
D5_LAT   :   0000 (OFF)
```

```
D6_LAT = (rotateReg & 4) >> 2;

rotateReg:   1000
4        : & 0100
---------------------
             0000

>> 2     :   0000

----------------------
D6_LAT   :   0000 (OFF)


D7_LAT = (rotateReg & 8) >> 3;

rotateReg:   1000
8        : & 1000
---------------------
             1000

>> 3     :   0001

----------------------
D7_LAT   :   0001 (ON)
```

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

Introduction

This lesson shows how to configure the ADC, run a conversion, read the analog voltage controlled by the potentiometer (POT1) on the board, and display the high order four bits on the display.

Hardware Effects

The top four MSBs of the ADC are displayed on the LEDs. Rotate the potentiometer to change the display.

Summary

Most PIC® devices have on-board Analog-to-Digital Converters (ADC) with resolution ranging from 8 to 16 bits. The converter can be referenced to the device’s V_{DD} or an external voltage reference. This lesson references it to V_{DD}. The result from the ADC is represented by a ratio of the voltage to the reference.

EQUATION 4-1:

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

Converting the answer from the ADC back to voltage requires solving for V.

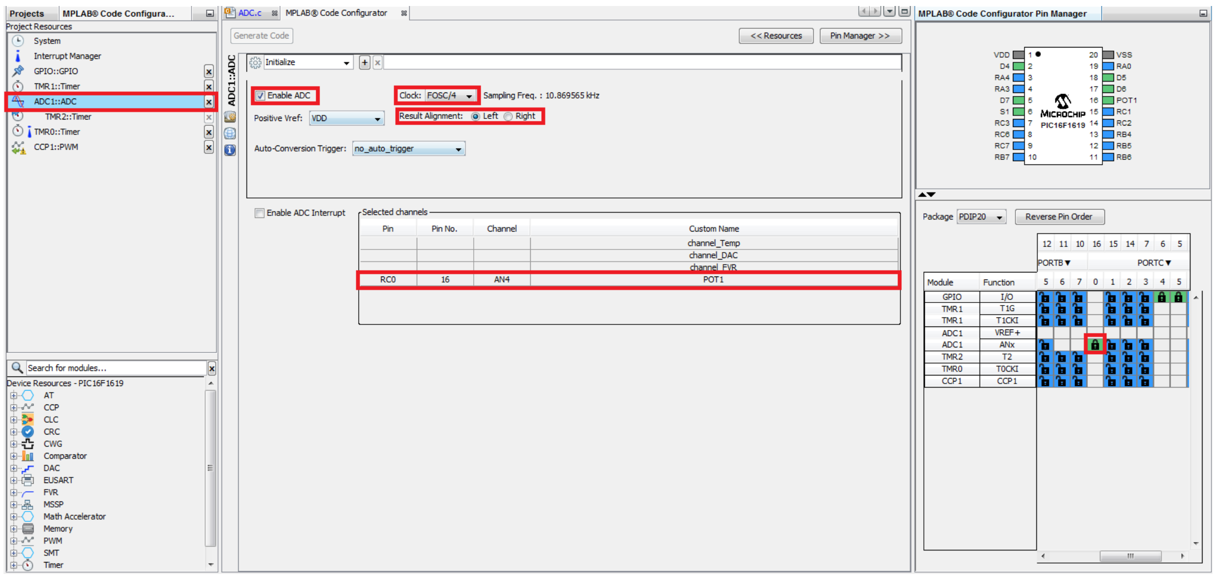
$$V = (ADC/1023)*V_{REF}$$

Here’s the checklist for this lesson:

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

A sample ADC configuration using MCC is shown below

FIGURE 4-1: MCC WINDOW – ADC MODULE



[✓] Enable ADC

Clock: FOSC/4 Sampling Freq. : 10.869565 kHz

Positive Vref: VDD

Result Alignment: [✓ Left] Right

Auto-Conversion Trigger: no_auto_trigger

[ ] Enable ADC Interrupt

Selected channels

Pin	Pin No.	Channel	Custom Name
			channel_Temp
			channel_DAC
			channel_FVR
RC0	16	AN4	POT1

		12	11	10	16	15	14	7	6	5
		PORTB ▼			PORTC ▼					
Module	Function	5	6	7	0	1	2	3	4	5
GPIO	I/O	🟦	🟦	🟦		🟦	🟦	🟦	🟩	🟩
TMR1	T1G	🟦	🟦	🟦		🟦	🟦	🟦		
TMR1	T1CKI	🟦	🟦	🟦		🟦	🟦	🟦		
ADC1	VREF+									
ADC1	ANx	🟦			🟩	🟦	🟦	🟦		
TMR2	T2	🟦	🟦	🟦		🟦	🟦	🟦		
TMR0	T0CKI	🟦	🟦	🟦		🟦	🟦	🟦		
CCP1	CCP1	🟦	🟦	🟦		🟦	🟦	🟦		

New Register

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. 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 as an analog input, the corresponding TRISx bit must be set to '1' to allow external control of the voltage on the pin.

This lesson sets RC0 as an analog input and uses the potentiometer connected to the said pin (POT1) to vary the voltage.

C Language

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

EXAMPLE 4.1: C CODE FOR "ADC" LAB

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

void ADC(void)
{
    //Get the top 4 MSBs and display it in the LEDs
    adcResult = ADC_GetConversion(POT1) >> 12;

    //Determine which LEDs will light up
    D4_LAT = adcResult & 1;
    D5_LAT = (adcResult & 2) >> 1;
    D6_LAT = (adcResult & 4) >> 2;
    D7_LAT = (adcResult & 8) >> 3;
}
```

```
adcResult = ADC_GetConversion(POT1) >> 12;

ADC_GetConversion(POT1) 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 `adcResult`, which is defined as "unsigned 16-bit integer" found in `ADC.h`. Then the bits of `adcResult` are shifted to the right by 12 places to that only the top 4 MSBs are left.

The following shows how the top 4 MSBs are extracted from the result of the conversion.

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>							
1	0	1	1	0	0	1	1	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 `0b10110011` and `ADRESL` is `0b11100101`.

After shifting:

adcResult

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

Shifting the value of `adcResult` 12 places to the right leaves us only with the top 4 MSBs which is `0b1011`.

```
D4_LAT = adcResult & 1;
D5_LAT = (adcResult & 2) >> 1;
D6_LAT = (adcResult & 4) >> 2;
D7_LAT = (adcResult & 8) >> 3;
```

These statements are used to reflect the value stored in `adcResult` onto the LEDs. The Bitwise AND operator is used to determine whether the LEDs output is high or low. Then the bits are shifted with respect to its position.

LESSON 5: VARIABLE SPEED ROTATE

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.

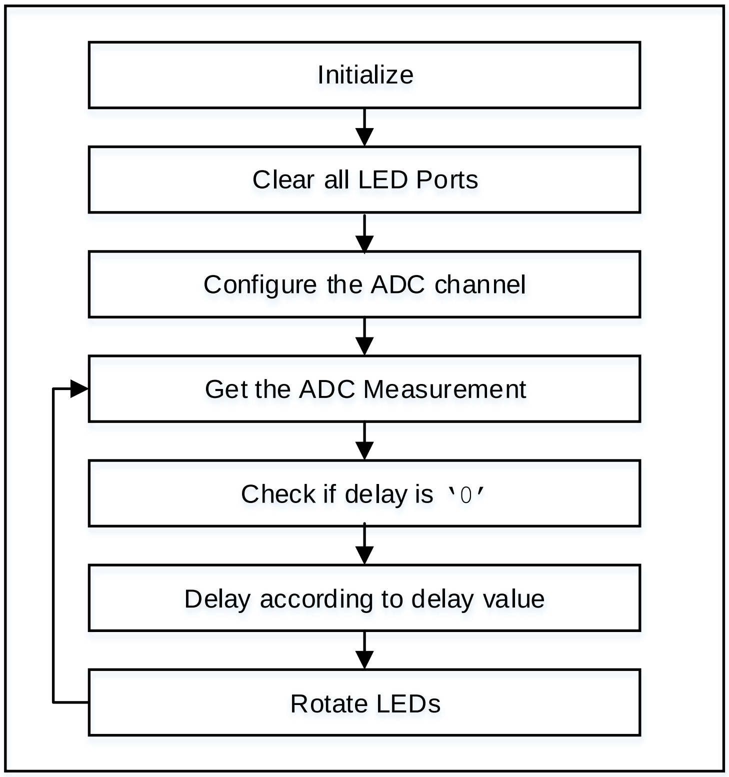
Hardware Effects

Rotate POT1 counterclockwise to see the LEDs shift faster.

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



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

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

#include "../../mcc_generated_files/mcc.h"
#include "../../global.h"

/*
                             Application
 */
void VSR(void)
{
    // Initialize temporary register to begin at 1
    rotateReg = 1;

    while(1){
        //Set DELAY from the top 8 MSbs of ADC
        delay =  ADC_GetConversion(POT1) >> 8;

        //Delay for at least 5ms
        __delay_ms(5);

        //Decrement the 8 MSbs of the ADC and delay 2ms for each
        while (delay-- != 0)
            __delay_ms(2);

        //Determine which LED will light up
        //i.e. which bit in the register the 1 has rotated to.
        D4_LAT = rotateReg & 1;
        D5_LAT = (rotateReg & 2) >> 1;
        D6_LAT = (rotateReg & 4) >> 2;
        D7_LAT = (rotateReg & 8) >> 3;

        //Rotate position of LED
        rotateReg = rotateReg << 1 ;

        //Return to initial position of LED
        if (rotateReg == 16)
            rotateReg = 1;
    }
}
/**
 End of File
 */
```

```
    //Set DELAY from the top 8 MSbs of ADC
    delay =  ADC_GetConversion(POT1) >> 8;
```

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

```
//Delay for at least 5ms
__delay_ms(5);

//Decrement the 8 MSbs of the ADC and delay 2ms for each
while (delay-- != 0)
     __delay_ms(2);
```

A minimum delay of 5 ms is set, and then the ‘`delay`’ variable is decremented until it reaches ‘`0`’. After which, another delay of 2 ms is set before the codes for rotation is executed.

LESSON 6: PULSE-WIDTH MODULATION (PWM)

Introduction

In this lesson, the PIC® MCU generates a PWM signal that lights an LED with the POT1 thereby controlling the brightness.

Hardware Effects

Rotating POT1 will adjust the brightness of a single LED D7.

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. A larger number of steps applied, which lengthens the pulse width, also supplies more power to the load. Lowering the number of steps applied, shortens the pulse width thus supplying less power. The PWM period is defined as the duration of one cycle or the total amount of on and off time combined.

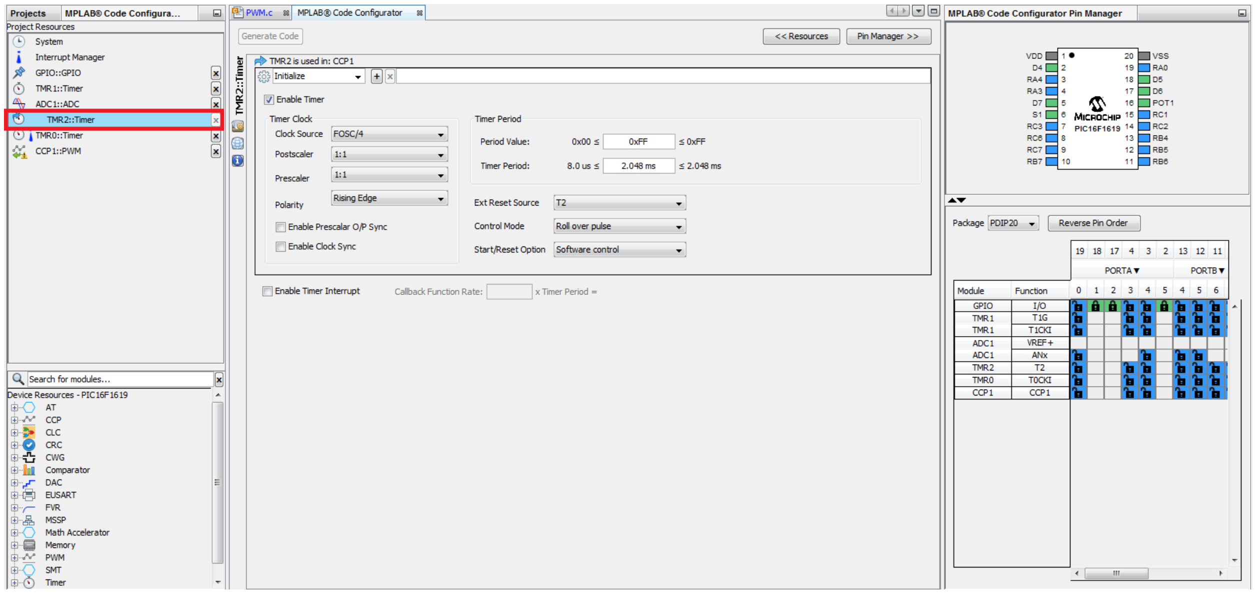
It is recommended that the reader refer to the Capture/Compare/PWM section in the data sheet to learn about related registers. This lesson will briefly cover how to setup a single PWM.

The PWM period is specified by the PRx register. Timer 2/4/6 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, but CCPRxH is read-only. When the timer is equal to PRx, 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

The following steps should be executed in the order shown when configuring the CCP module using MCC for standard PWM operation:

FIGURE 6-1: MCC WINDOW – TMR2 MODULE



TMR2::Timer

TMR2 is used in: CCP1

Initialize

[✓] Enable Timer

Timer Clock
- Clock Source: FOSC/4
- Postscaler: 1:1
- Prescaler: 1:1
- Polarity: Rising Edge
- [ ] Enable Prescalar O/P Sync
- [ ] Enable Clock Sync

Timer Period
- Period Value: 0x00 ≤ 0xFF ≤ 0xFF
- Timer Period: 8.0 us ≤ 2.048 ms ≤ 2.048 ms

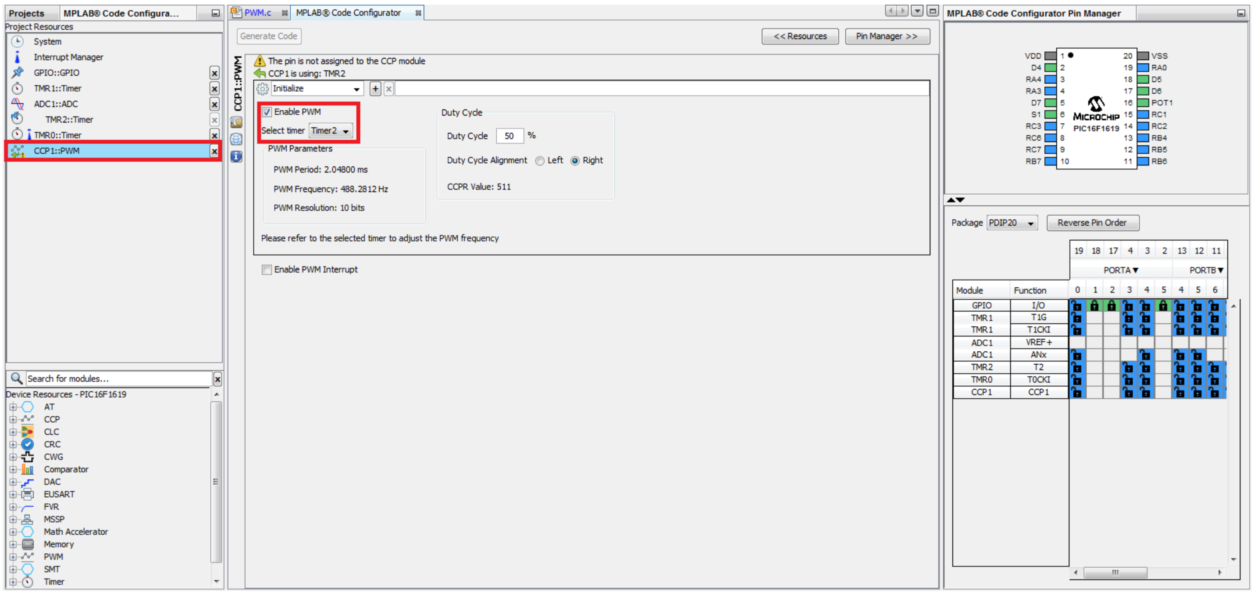
Ext Reset Source: T2

Control Mode: Roll over pulse

Start/Reset Option: Software control

[ ] Enable Timer Interrupt   Callback Function Rate: x Timer Period =

FIGURE 6-2: MCC WINDOW – CCP1:PWM MODULE



CCP1::PWM

The pin is not assigned to the CCP module

CCP1 is using: TMR2

Initialize

[✓] Enable PWM

Select timer: Timer2

PWM Parameters
- PWM Period: 2.04800 ms
- PWM Frequency: 488.2812 Hz
- PWM Resolution: 10 bits

Duty Cycle
- Duty Cycle: 50 %
- Duty Cycle Alignment: Left [✓ Right]
- CCPR Value: 511

Please refer to the selected timer to adjust the PWM frequency

[ ] Enable PWM Interrupt

EQUATION 6-1: Computing for PWM resolution

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

Two conditions must hold true for this lesson:
1. 10 bits of resolution
2. No flicker in LED

C Language

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

EXAMPLE 6.1: C CODE FOR “PWM” LAB

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

void PWM(void)
{
    // Start Timer2
    TMR2_StartTimer();

    // Set D7 as output of CCP1 using PPS
    RC5PPS = 0b00001100;

    While(1){
        //Start ADC conversion
        ADC_StartConversion(POT1);

        //Wait for the conversion to finish
        while(ADC_IsConversionDone());

        // Store the top 8 MSbs into CCPR1L
        CCPR1L = ADRESH;

         // Store the 2 LSbs into CCP1CON[5:4] to complete the 10bit
         // resolution
        CCP1CONbits.DC1B = ADRESL >> 6;
    }
}
```

```
TMR2_Start();
```

This function simply starts the Timer2 module of the PIC® MCU by setting the `TMR2ON` bit of the `T2CON` register.

```
RC5PPS = 0b00001100;
```

This statement sets RC5 as the output pin of the CCP1 module.

```
ADC_StartConversion(POT1);
while(ADC_IsConversionDone());

// Store the top 8 MSbs into CCPR1L
CCPR1L = ADRESH;

// Store the 2 LSbs into CCP1CON[5:4] to complete the 10bit
// resolution
CCP1CONbits.DC1B = ADRESL >> 6;
```

The statements above start the ADC module and wait for the conversion to be finished. Once the conversion is done, the top 8 MSBs from `ADRESH` are stored into `CCPR1L`, while the 2 LSBs are stored into `CCPR1H` register to complete the 10bit resolution.

```
RC5PPS = 0b00000000;
```

This restores RC5 as a normal output pin.

LESSON 7: TIMER1

Introduction

This lesson will produce the same output as Lesson 3: Rotate. The only difference is that this version uses Timer1 to provide the delay routine.

Hardware Effects

LEDs rotate from right to left, similar to Lesson 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 method of measuring time or delay loops as it allows the processor to work on other tasks rather than counting instruction cycles.

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

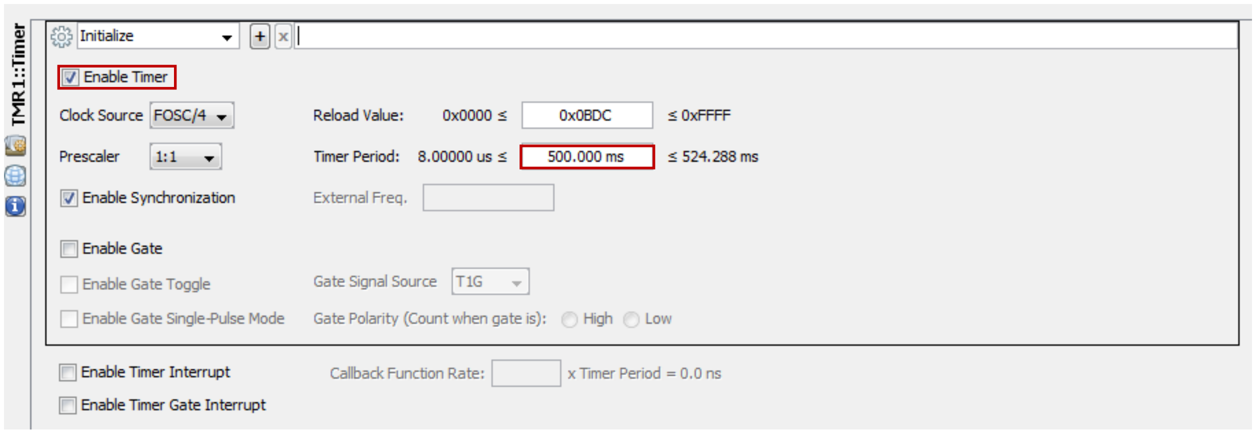
`TMR1H` and `TMR1L` are 8-bit registers that form a 16-bit timer/counter register pair. This timer/counter increments from a defined value until it reaches a value of `255` or `0xFF` each, and overflows. An overflow will set the Timer1 flag bit 'high' and trigger an interrupt when enabled.

PIR1

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

A sample Timer1 configuration using MCC is shown below. Timer1 is configured to overflow every 500 ms.

FIGURE 7-1: MCC COMPOSER AREA – TMR1 MODULE



☑ Enable Timer

Clock Source FOSC/4 Reload Value: 0x0000 ≤ 0x0BDC ≤ 0xFFFF

Prescaler 1:1 Timer Period: 8.00000 us ≤ 500.000 ms ≤ 524.288 ms

☑ Enable Synchronization External Freq.

☐ Enable Gate

☐ Enable Gate Toggle Gate Signal Source T1G

☐ Enable Gate Single-Pulse Mode Gate Polarity (Count when gate is): ○ High ○ Low

☐ Enable Timer Interrupt Callback Function Rate: x Timer Period = 0.0 ns

☐ Enable Timer Gate Interrupt

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

EXAMPLE 7.1: INITIALIZING TIMER1

`TMR1_Initialize();`
Before Instruction:
All registers/bits related to Timer1 are disabled or set to default.
After Instruction:
Registers/bits and variables related to Timer1 are enabled or set according to the user’s input in the MCC. These include:
`T1CON`,
`T1GCON`,
`TMR1H`,
`TMR1L`,
`PIR1bits.TMR1IF`,
and `timer1ReloadVal`.
`TMR1_StartTimer();` instruction is also called.

EXAMPLE 7.2: STARTING TIMER1

`TMR1_StartTimer();`
Before Instruction:
`T1CONbits.TMR1ON = 0;`
After Instruction:
`T1CONbits.TMR1ON = 1;`

EXAMPLE 7.3: STOPPING TIMER1

```
TMR1_StopTimer();
```
Before Instruction:
```
T1CONbits.TMR1ON = 1;
```
After Instruction:
```
T1CONbits.TMR1ON = 0;
```

EXAMPLE 7.4: RELOADING TIMER1

```
TMR1_Reload();
```
Before Instruction:
```
TMR1H = 0;
TMR1L = 0;
```
After Instruction:
```
TMR1H = (timer1ReloadVal >> 8);
TMR1L = timer1ReloadVal;
```

C Language

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

Example 7.5: C CODE FOR “TIMER1” LAB

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

#include "../../mcc_generated_files/mcc.h"
#include "../../global.h"

/*
                             Application
 */
void Timer1(void)
{
    // Start Timer1
    TMR1_StartTimer();

    // Begin with D4 High
    D4_LAT = 1;

    // Initialize temporary register to begin at 1
    rotateReg = 1;

    while(1){
        //Wait for Timer1 to overflow
        while(!TMR1IF);

         //Reload the initial value of TMR1
        TMR1_Reload();

        //Rotate position of LED
        rotateReg = rotateReg<<1;
        //Return to initial position of LED
        if (rotateReg == 16)
            rotateReg = 1;

        //Determine which LED will light up
        //i.e. which bit in the register the 1 has rotated to.
        D4_LAT = rotateReg & 1;
        D5_LAT = (rotateReg & 2) >> 1;
        D6_LAT = (rotateReg & 4) >> 2;
        D7_LAT = (rotateReg & 8) >> 3;

        //Clear the TMR1 interrupt flag
        TMR1IF = 0;
    }
    // Stop Timer1
    TMR1_StopTimer();
}

/**
 End of File
 */
```

```
TMR1_StartTimer();
```

Equivalent:

```
void TMR1_StartTimer(void)
{
    // Start the Timer by writing to TMRxON bit
    T1CONbits.TMR1ON = 1;
}
```

This function simply starts the Timer1 module of the PIC® MCU by setting the '`TMR1ON`' bit of the '`T1CON`' register.

```
//Wait for Timer1 to overflow
while(!TMR1IF);
```

This statement waits for the Timer1 to overflow and its corresponding flag to set .

```
TMR1_Reload();
```

Equivalent:

```
void TMR1_Reload(void)
{
    //Write to the Timer1 register
    TMR1H = (timer1ReloadVal >> 8);
    TMR1L = timer1ReloadVal;
}
```

As '`TMR1IF`' bit is set, '`TMR1H`' and '`TMR1L`' are cleared. These registers need to reload the value determined by '`timer1ReloadVal`' at '`TMR1_Initialize()`' for the delay to be consistent.

At initialization:

```
TMR1H = 0x0B; TMR1L = 0xDC; //500ms with Prescaler of 1:1
timer1ReloadVal=(TMR1H << 8) | TMR1L;
```

TMR1H <15:8>								TMR1L <7:0>							
0	0	0	0	1	0	1	1	1	1	0	1	1	1	0	0

```
TMR1IF = 0;
```

'`TMR1IF`' bit is then cleared for the next cycle of Timer1.

```
TMR1_StopTimer();
```

Equivalent:

```
void TMR1_StopTimer(void)
{
    // Stop the Timer by writing to TMRxON bit
    T1CONbits.TMR1ON = 0;
}
```

This disables the use of Timer1 for the next labs.

LESSON 8: INTERRUPTS

Introduction

This lesson discusses about interrupts – its purpose, capabilities and how to set them up. Most interrupts are sourced from MCU peripheral modules. Some I/O pins can also be configured to generate interrupts when they change state. Interrupts usually signal events that require servicing by the software’s Interrupt Service Routine (ISR). Once an interrupt occurs, the program counter immediately jumps to the ISR and executed all its instructions before exiting. After an exit from the ISR, the program counter jumps to the instruction right after the one executed before the interrupt occurred. It is a rather more efficient way of watching out for events than continuously polling a bit or register.

Hardware Effects

D4, D5, D6 and D7 rotate from right to left at a constant rate of 500 ms.

Summary

This lab demonstrates the advantage of using interrupts over polling. An interrupt is generated whenever the Timer0 register reaches `0xFF` and goes back to reset value. This indicates that 500 ms has passed and it is time to rotate the light. This interrupt is serviced by the `TMR0_ISR()` function. Note that this is the same for Lab 7 but this time, we are not continuously polling the `TMR1IF` flag.

New Register

Register	Purpose
INTCON	Contains the various enable and flag bits for the usual interrupt sources.

Note: INTCON register bit assignments vary from device to device. Please check the datasheet of your device for more details.

Below is the INTCON register for PIC16F1708.

INTCON Register <7:0>							
GIE	PEIE	TMR0IE	INTE	IOCIE	TMR0IF	INTF	IOCIF

- Bit 7 : GIE – Global Interrupt Enable Bit
- Bit 6 : PEIE – Peripheral Interrupt Enable Bit
- Bit 5 : TMR0IE – Timer0 Interrupt Enable Bit
- Bit 4 : INTE – INT External Interrupt Enable Bit
- Bit 3 : IOCIE – Interrupt-on-change Enable Bit
- Bit 2 : TMR0IF – Timer0 Overflow Interrupt Flag Bit
- Bit 1 : INTF – INT External Interrupt Flag Bit
- Bit 0 : IOCIF – Interrupt-on-change Flag Bit

C Language

The codes below demonstrate how to set up interrupts for Timer0 peripheral. Please note that different peripherals have different set-up procedures. This can be taken care of by the MCC for you. Please refer to the datasheet of your device if you wish to set them up manually.

Main Program and Set-up

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

void Interrupt(void)
{
    //Enable the Global Interrupts
    INTERRUPT_GlobalInterruptEnable();

    //Enable the Peripheral Interrupts
    INTERRUPT_PeripheralInterruptEnable();

    //Enable the TMR0 Interrupts
    TMR0IE = 1;

    rotateReg = 1;

    //Wait for interrupt to occur
    while(1)
        {
            ;
        }
}
```

The following 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.

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

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 would proceed to the function designated to handle the interrupt.

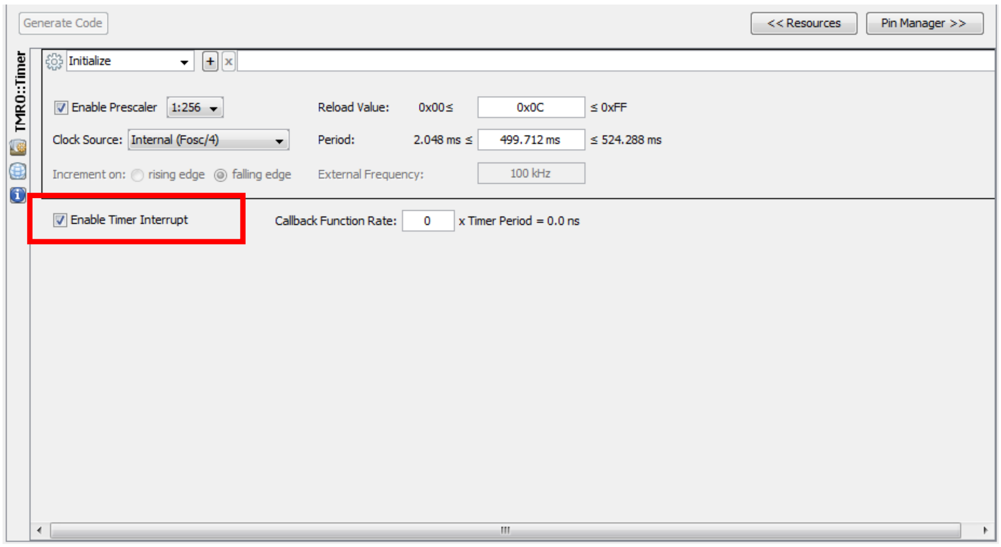
```
#include "interrupt_manager.h"
#include "mcc.h"

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

Timer0 Overflow Interrupt Handler (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) {
    //If the last LED has been lit, restart the pattern
    if(rotateReg == 1)
        rotateReg = 16;

    //Shift value in temp register to the left by 1 bit
    rotateReg >>= 1;

    //Check which LED should be lit
    D4_LAT = rotateReg & 1;
    D5_LAT = (rotateReg & 2) >> 1;
    D6_LAT = (rotateReg & 4) >> 2;
    D7_LAT = (rotateReg & 8) >> 3;

    //Clear the TMR0 interrupt flag. Don't forget!
    INTCONbits.TMR0IF = 0;

    //Reload the initial value of TMR0
    TMR0 = timer0ReloadVal;
}
```

FIGURE 8-1: MCC COMPOSER AREA FOR TIMER0 MODULE WITH INTERRUPTS



✓ Enable Prescaler 1:256
Reload Value: 0x00≤ 0x0C ≤ 0xFF
Clock Source: Internal (Fosc/4)
Period: 2.048 ms ≤ 499.712 ms ≤ 524.288 ms
Increment on: rising edge falling edge
External Frequency: 100 kHz
✓ Enable Timer Interrupt
Callback Function Rate: 0 x Timer Period = 0.0 ns

LESSON 9: WAKE-UP FROM SLEEP USING WATCHDOG TIMER

Introduction

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

Hardware Effects

Once this lab is set to `RUNNING` state, the watchdog timer will start counting. D4 and D6 are ON while the MCU is in sleep mode. Pressing the switch during Sleep will have no effect on the program won't go to the next lab since the PIC® is in Sleep mode. After the watchdog timer reached its period, (for this lab is approximately 16 seconds) the PIC® exits sleep mode and the LEDs are toggled.

Summary

The Power-Down mode is entered by executing the `SLEEP` instruction. Upon entering Sleep mode, there are different conditions that exist such as:

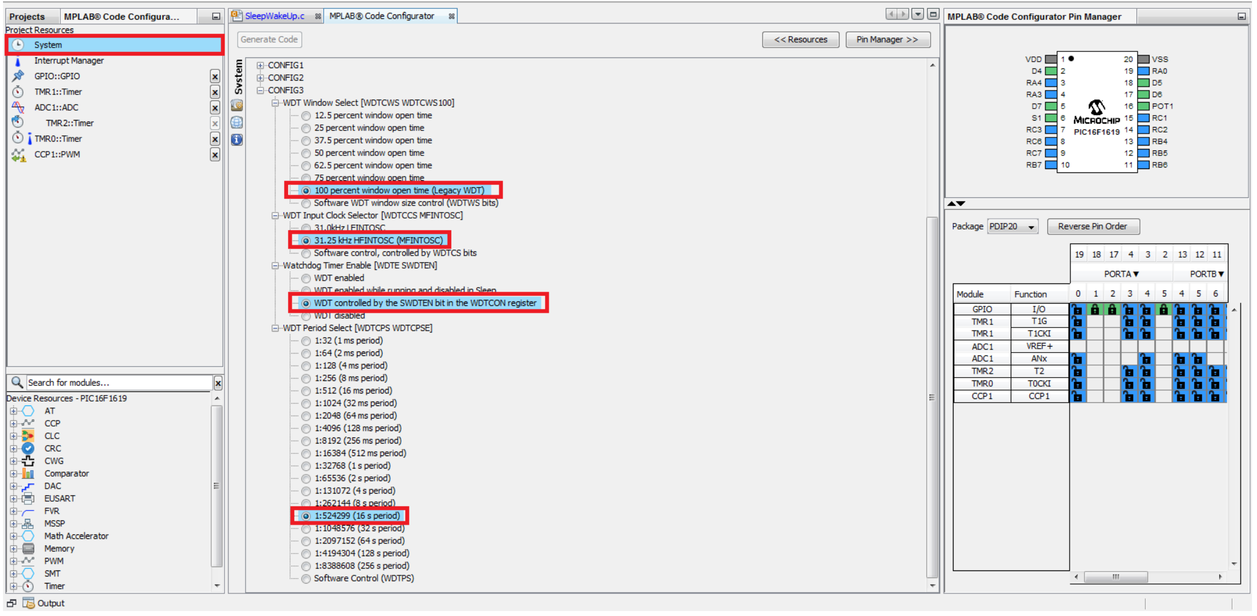
- WDT will be cleared but will keep on counting, if enabled for operation during Sleep.
- PD bit of the STATUS register is cleared.
- TO bit of the STATUS register is set.
- CPU clock is disabled.

Effects of Sleep may vary per device. It is recommended that the user refer to the specific device datasheet for detailed information on Sleep.

The Watchdog Timer (WDT) is a system timer that generates a Reset if the firmware does not issue a `CLRWDT` instruction within the time-out period. WDT is typically used to recover the system from unexpected events. When the device enters Sleep, the WDT is cleared. If the WDT is enabled during Sleep, the WDT resumes counting. When the device exits Sleep, the WDT is cleared again. When a WDT time-out occurs while the device is in Sleep, no Reset is generated.

The following steps should be executed to configure the watchdog timer using the MCC.

FIGURE 9-1: MCC WINDOW – SYSTEM CONFIGURATIONS



```
CONFIG1
CONFIG2
CONFIG3
  WDT Window Select [WDTCWS WDTCWS100]
    ( ) 12.5 percent window open time
    ( ) 25 percent window open time
    ( ) 37.5 percent window open time
    ( ) 50 percent window open time
    ( ) 62.5 percent window open time
    ( ) 75 percent window open time
    (•) 100 percent window open time (Legacy WDT)
    ( ) Software WDT window size control (WDTWS bits)
  WDT Input Clock Selector [WDTCCS MFINTOSC]
    ( ) 31.0kHz LFINTOSC
    (•) 31.25 kHz HFINTOSC (MFINTOSC)
    ( ) Software control, controlled by WDTCS bits
  Watchdog Timer Enable [WDTE SWDTEN]
    ( ) WDT enabled
    ( ) WDT enabled while running and disabled in Sleep
    (•) WDT controlled by the SWDTEN bit in the WDTCON register
    ( ) WDT disabled
  WDT Period Select [WDTCPS WDTCPSE]
    ( ) 1:32 (1 ms period)
    ( ) 1:64 (2 ms period)
    ( ) 1:128 (4 ms period)
    ( ) 1:256 (8 ms period)
    ( ) 1:512 (16 ms period)
    ( ) 1:1024 (32 ms period)
    ( ) 1:2048 (64 ms period)
    ( ) 1:4096 (128 ms period)
    ( ) 1:8192 (256 ms period)
    ( ) 1:16384 (512 ms period)
    ( ) 1:32768 (1 s period)
    ( ) 1:65536 (2 s period)
    ( ) 1:131072 (4 s period)
    ( ) 1:262144 (8 s period)
    (•) 1:524299 (16 s period)
    ( ) 1:1048576 (32 s period)
    ( ) 1:2097152 (64 s period)
    ( ) 1:4194304 (128 s period)
    ( ) 1:8388608 (256 s period)
    ( ) Software Control (WDTPS)
```

C Language

A sample code written in C language for the "Sleep Wake-Up" lab is provided below.

EXAMPLE 9.1: C CODE FOR "SLEEP WAKE-UP" LAB

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

void SleepWakeUp(void)
{
    // Turn ON D4 and D6
    D4_LAT = D6_LAT = 1;

    // Turn OF D5 and D7
    D5_LAT = D7_LAT = 0;

    // Set the WDT period to 16s
    WDTCONbits.WDTPS = 0b01110;

    // Enable Watchdog Timer
    WDTCONbits.SWDTEN = 1;

    // Enter SLEEP MODE
    SLEEP();

    while(1){
        //Wait 16 seconds for the WDT time out and then turn OFF D4 and D6
        //and turn ON D5 and D7
        D4_LAT = D6_LAT = 0;
        D5_LAT = D7_LAT = 1;

        //Disable Watchdog Timer
```

```
          WDTCON0bits.SEN = 0;
     }
}
```

```
SLEEP();
```

This function puts the device into Sleep mode.

LESSON 10: EEPROM

Introduction

This lesson provides code for writing and reading a single byte onto the on-board EEPROM. EEPROM is nonvolatile memory, meaning that it does not lose its value when power is shut off. This is unlike RAM, loses its value when no power is applied. The EEPROM is useful for storing variables that must still be present during no power condition. 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. A PIC16F1829 device having 256 bytes of EEPROM is used in the following example.

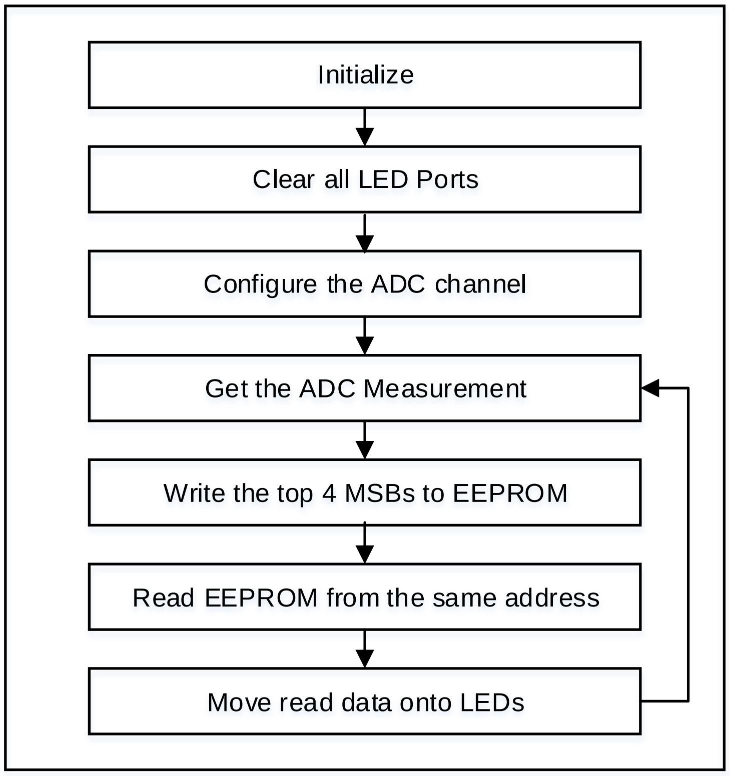
Hardware Effects

The top 4 MSBs of the ADC is written to EEPROM. These are read afterwards and displayed on the LEDs. Rotating POT1 changes the ADC value to be stored into and read from EEPROM.

Summary

This lab is almost similar to **LESSON 4: ADC** the difference being the ADC result displayed on the LEDs is written to and read from the EEPROM instead of directly reading the ADRES registers. As shown on **FIGURE 10-1** below, the top 4 MSBs of the ADC result is first written to EEPROM, and retrieved later from the same address before moving onto the LEDs. This is done by issuing the `eeprom_write()` and `eeprom_read()` instructions respectively.

FIGURE 10-1: PROGRAM FLOW


Initialize
Clear all LED Ports
Configure the ADC channel
Get the ADC Measurement
Write the top 4 MSBs to EEPROM
Read EEPROM from the same address
Move read data onto LEDs

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 the 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.

Instructions

Instruction	Purpose
`eeprom_write(unsigned char addr, unsigned char value)`	Writes to EEPROM data memory address `addr` the value stored in variable `value`
`eeprom_read(unsigned char addr)`	Reads from EEPROM address `addr`

The instructions above are functions that call the macros `EEPROM_WRITE()` and `EEPROM_READ()` respectively. These are included in the MPLAB® XC8 compiler library and are used to configure the different EEPROM registers automatically.

C Language

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

Example 10.1: C CODE FOR "EEPROM" LAB

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

#include "../../mcc_generated_files/mcc.h"
#include "../../global.h"

/*
                              Application
 */
void EEPROM(void){

    //Get the top 4 MSBs of the ADC and write them to EEPROM
    adcResult = ADC_GetConversion(POT1) >> 12;
    eeprom_write(0x00, adcResult);

    //Load whatever is in EEPROM to the LATCH
    ledDisplay = eeprom_read(0x00);

    //Determine which LEDs will light up
    D4_LAT = ledDisplay & 1;
    D5_LAT = (ledDisplay & 2) >> 1;
    D6_LAT = (ledDisplay & 4) >> 2;
    D7_LAT = (ledDisplay & 8) >> 3;
}

/*
 End of File
 */
```

```
eeprom_write(0x00, adcResult);
```

This function writes to the program memory at address ‘`0x00`’ the values stored within ‘`adcResult`’ (similar to **LESSON 4**).

```
ledDisplay = eeprom_read(0x00);
```

The function above reads the EEPROM data located at address ‘`0x00`’ and then stores the read data in a user-defined global variable ‘`ledDisplay`’. This data will be reflected on the LED ports as seen in the codes following `eeprom_read()`.

LESSON 11: HIGH-ENDURANCE FLASH MEMORY

Introduction

In this lesson, we will discuss High-Endurance Flash (HEF) Memory, an alternative to Data EEPROM memory present in many devices. Most new devices have both types of memory but others have only one or the other. 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.

Hardware Effects

D4 and D6 will light up as we write '`5`' into the HEF memory of the device.

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 form the Flash memory array. This is in contrast to true data EEPROMs which do not stall MCU executions during a write cycle.

C Language

The following code is the core of the lab. It just uses functions `HEFLASH_writeBlock` and `HEFLASH_readBlock` to write and then read back the value stored. These functions are part of a library for HEF functions written for the application note *AN1673: Using the PIC16F1XXX High-Endurance Flash (HEF) Block*. For the specifics on the code library, you can download a copy of the said application note at the Microchip website.

```
//Store your data here
char buffer[];

//Write value 0b0101 to buffer register
buffer[0] = 5;

//Write contents of buffer to 1 byte of Block 1 of HEF memory. Ensure success.
errorCheck = HEFLASH_writeBlock( 1, (void*)&buffer, 1);
assert (errorCheck == 0);

//Read back value stored
errorCheck = HEFLASH_readBlock((void*)&buffer, 1, 1);
assert ( errorCheck == 0);

//Display on LEDs
rotateReg = *buffer;
```

```
D4_LAT = rotateReg & 1;
D5_LAT = (rotateReg & 2) >> 1;
D6_LAT = (rotateReg & 4) >> 2;
D7_LAT = (rotateReg & 8) >> 3;
```