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 (or 11 for some devices) lab exercises that demonstrate the myriad basic capabilities of PIC® devices and can also be used to test the condition 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.

LESSONS

The lessons in this document are presented in the same order as they appear on the programmed labs. You can progress through each of the labs by simply pressing the S1 button of your board.

- **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: Pulse-Width Modulation (PWM)**
- **Lesson 7: Timer1**
- **Lesson 8: Interrupts**
- **Lesson 9: Wake-up from Sleep Using Watchdog Timer**
- **Lesson 10: EEPROM**(1)
- **Lesson 11: 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.

INPUTS AND DISPLAY

- **Push Button Switch** – One push button switch S1 is provided on the board. S1 is connected to the PIC MCU's RC4 pin and is used to switch to the next lab.
- **Potentiometer** – A 10kΩ potentiometer POT1 is used in labs requiring analog inputs.
- **LEDs** - The Curiosity Development Board has four red LEDs (D7 through D4) that are connected to I/O ports RC5, RA2, RA1 and RA5, respectively. These LEDs are used to display the output of the different labs.

LESSON 1: HELLO WORLD (TURN ON AN LED)

Introduction

The first lesson shows how to turn on an LED.

Hardware Effects

LED 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 (LED_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. Writes should be performed on the LAT register instead on the port directly.

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 programmer 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
`LED_D4_SetHigh()`	Make the bit value of LED_D4 (LATA5) a ‘1’ (5V)
`LED_D4_SetLow()`	Make the bit value of LEDD4 (LATA5) a ‘0’ (0V)

EXAMPLE 1.1: SETTING A BIT INTO ‘1’

```
LED_D4_SetHigh();
```

Before Instruction:

```
LATA5 = 0;
```

After Instruction:

```
LATA5 = 1;
```

EXAMPLE 1.2: SETTING A BIT INTO ‘0’

```
LED_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

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

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

/*
                             Application
 */

void HelloWorld(void) {

    if (labState != RUNNING) {
```

```
        LED_D4_LAT = LED_D5_LAT = LED_D6_LAT = LED_D7_LAT = OFF;

        labState = RUNNING;
    }

    if (labState == RUNNING) {
        LED_D4_SetHigh();
    }

    //Check if a switch event occurs
    if (switchEvent) {
        labState = NOT_RUNNING;
    }
}
/**
 End of File
 */
```

```
//
```

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

```
#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 "../../labHeader.h"
```

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

```
if (labState != RUNNING) {
    LED_D4_LAT = LED_D5_LAT = LED_D6_LAT = LED_D7_LAT = OFF;
    labState = RUNNING;
}
```

This statement checks whether the *HelloWorld (Lab 01)* is running or not. If the state of this lab is currently `NOT_RUNNING`, then the code above will clear all the LED PORTs and change the state of the lab to `RUNNING`.

```
if (labState == RUNNING) {
    LED_D4_SetHigh();
}
```

This statement calls the function `LED_D4_SetHigh()` if the state of the lab is `RUNNING`. `LED_D4_SetHigh()` turns on LED D4.

```
LED_D4_SetHigh();
```

Equivalent:

```
#define LED_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 making it "high".

```
if (switchEvent) {
    labState = NOT_RUNNING;
}
```

This statement checks if the S1 button is pressed. If the button is pressed (`switchEvent = 1`), the state of the lab will be changed to `NOT_RUNNING`.

LESSON 2: BLINK

Introduction

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

Hardware Effects

LED 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 '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 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
`LED_D4_Toggle()`	Changes the bit value of LED_D4 (LATA5) from '`0`' to '`1`', or '`1`' to '`0`'

EXAMPLE 2.1: TOGGLING A BIT

```
LED 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/pin_manager.h"
#include "../../mcc_generated_files/tmr1.h"
#include "../../labHeader.h"

/*
                             Application
 */

#define OVERFLOW    3

void Blink(void) {

    static uint8_t counter;

    if (labState != RUNNING) {
        TMR1_StartTimer();

        LED_D4_LAT = LED_D5_LAT = LED_D6_LAT = LED_D7_LAT = OFF;

        labState = RUNNING;
    }

    if (labState == RUNNING) {
        //Wait for Timer1 to overflow
        while (!TMR1IF);

        TMR1_Reload();

        counter++;

        //Wait for overflow for 1.5 secs delay
        if (counter == OVERFLOW) {
            LED_D4_Toggle();
            counter = 0;
        }

        //Clear TMR1 Overflow flag
        TMR1IF = 0;
    }

    //Check if a switch event occurs
    if (switchEvent) {
        TMR1_StopTimer();

        labState = NOT_RUNNING;
    }
```

```
}
/**
 End of File
 */
```

```
#define OVERFLOW    3
```

A variable 'OVERFLOW' is defined with a constant decimal value of '3'.

```
static uint8_t counter;
```

A static variable 'counter' variable is declared.

```
if (labState != RUNNING) {
    TMR1_StartTimer();

    LED_D4_LAT = LED_D5_LAT = LED_D6_LAT = LED_D7_LAT = OFF;

    labState = RUNNING;
}
```

The MCC-generated macro TMR1_StartTimer() is used to start Timer1 and all LEDs are initially turned off. If the state of the lab is RUNNING, 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: TIMER1**).

```
if (labState == RUNNING) {
    //Wait for Timer1 to overflow
    while (!TMR1IF);

    TMR1_Reload();

    counter++;

    //Wait for overflow for 1.5 secs delay
    if (counter == OVERFLOW) {
        LED_D4_Toggle();
        counter = 0;
    }

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

The static variable 'counter' increments every time 'TMR1IF' is set until it reaches a value equal to 'OVERFLOW' which is previously defined as a variable having a constant value of '3'. This signifies that Timer1 has overflowed after 500 ms three times for a total of 1.5 secs, before LED D4 is toggled. 'counter' is then reset to '0' and the process is repeated.

```
LED_D4_Toggle();
```

Equivalent:

```
#define LED_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, 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 before instruction:

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

STATUS register before 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 before instruction:

<7:0>							
1	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	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.

STATUS Register <7:0>							
IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C

- Bit 7 : IRP – Register Bank Select Bit
- Bit 6-5 : RP<1:0> - Register Bank Select
- Bit 4: $\overline{TO}$ – Time Out bit
- Bit 3: $\overline{PD}$ – Power Down bit
- Bit 2: Z – Zero bit
- Bit 1: DC – Digit Carry bit
- Bit 0: C – Carry bit

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 "../../labHeader.h"

/*
                             Application
 */
```

```
#define LAST    16

static uint8_t rotateReg;

void Rotate(void) {

    if (labState != RUNNING) {
        LED_D4_LAT = ON;
        LED_D5_LAT = LED_D6_LAT = LED_D7_LAT = OFF;

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

        labState = RUNNING;
    }

    if (labState == RUNNING) {

        Delay500ms();

        rotateReg <<= 1;

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

        //Determine which LED will light up
        //ie. which bit in the register the 1 has rotated to.
        LED_D4_LAT = rotateReg & 1;
        LED_D5_LAT = (rotateReg & 2) >> 1;
        LED_D6_LAT = (rotateReg & 4) >> 2;
        LED_D7_LAT = (rotateReg & 8) >> 3;
    }

    //Check if a switch event occurs
    if (switchEvent) {
        labState = NOT_RUNNING;
    }
}

//Delay function to keep the LED on for 0.5 secs before rotating

void Delay500ms(void) {
    uint8_t 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 how the bitwise AND operation reflects the value of `rotateReg` (0b1000 in this example) onto 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 on-board potentiometer (POT1), and display the high order four bits on the display.

Hardware Effects

The top four MSBs of the ADC are reflected onto the LEDs. Rotate the potentiometer to change the display.

Summary

The PIC devices have an on-board Analog-to-Digital Converter (ADC) with 10 bits of resolution on any of 12 the channels available (*Note: The resolution and channels vary amongst the devices. Refer to the datasheet.*).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 WITH 10-BIT RESOLUTION

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

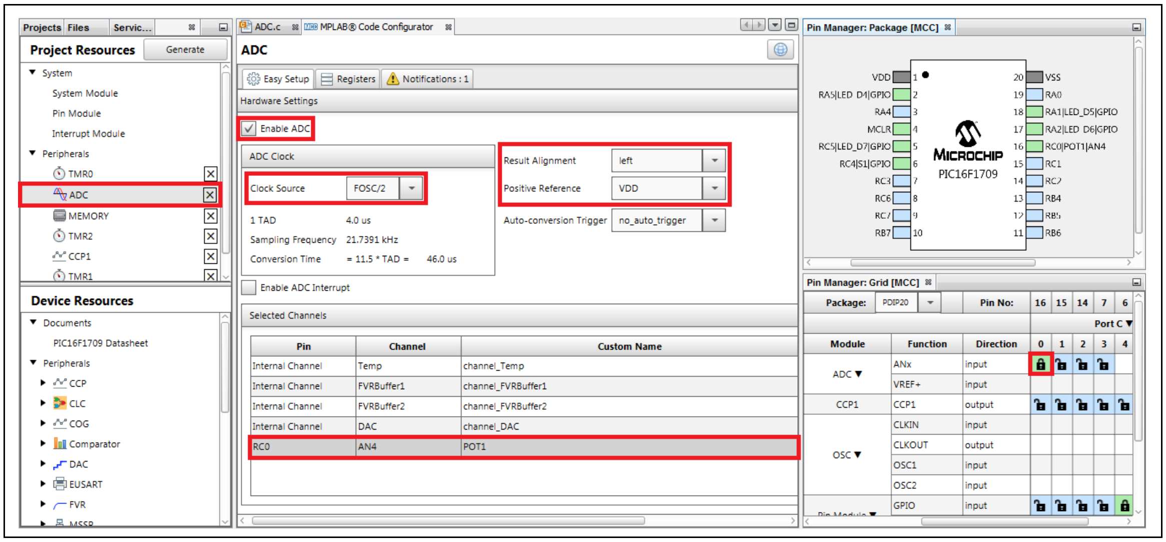
Converting the answer from the ADC back to voltage requires solving for V.

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

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.

FIGURE 4-1: MCC WINDOW – ADC MODULE



New Register

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

ANSELx

The ANSELx register determines whether the pin is a digital (1 or 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 to an analog input, the corresponding TRISx bit must be set to input mode in order to allow external control of the voltage on the pin.

This lesson sets RC0 as an analog input since the potentiometer (POT1) will 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

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

#include "../../mcc_generated_files/pin_manager.h"
#include "../../mcc_generated_files/adc.h"
#include "../../labHeader.h"

/*
                              Application
 */
```

```
static uint8_t adcResult;

void ADC(void) {

    if (labState != RUNNING) {
        LED_D4_LAT = LED_D5_LAT = LED_D6_LAT = LED_D7_LAT = OFF;

        labState = RUNNING;
    }

    if (labState == RUNNING) {
        //Get the top 4 MSBs and display it on the LEDs
        adcResult = ADC_GetConversion(POT1) >> 12;

        //Determine which LEDs will light up
        LED_D4_LAT = adcResult & 1;
        LED_D5_LAT = (adcResult & 2) >> 1;
        LED_D6_LAT = (adcResult & 4) >> 2;
        LED_D7_LAT = (adcResult & 8) >> 3;
    }

    //Check if a switch event occurs
    if (switchEvent) {
        labState = NOT_RUNNING;
    }
}
/**
 End of File
 */
```

```
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 acquisition time delay, starts the conversion, and returns the result of the conversion. The result of the conversion is stored in the `adc_result_t`, which is defined as "unsigned 16-bit integer" in `adc.h`. Then the bits of the `adcResult` are shifted to the right by 12 places so 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:

`adc_result_t`

<15:8>								<7:0>							

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

After conversion:

`adc_result_t`

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 `adc_result_t`. 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`.

```
LED_D4_LAT = adcResult & 1;
LED_D5_LAT = (adcResult & 2) >> 1;
LED_LAT = (adcResult & 4) >> 2;
LED_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. The following shows the bitwise AND operation on how the value of `adcResult (1011)` is reflected to the LEDs.

```
LED_D4_LAT = adcResult & 1;

adcResult:   1011
1        : & 0011
----------------------
LED_D4_LAT   :   0001 (ON)

LED_D5_LAT = (adcResult & 2) >> 1;

adcResult:   1011
2        : & 0010
---------------------
             0010

>> 1     :   0001

----------------------
LED_D5_LAT   :   0001 (ON)


LED_D6_LAT = (adcResult & 4) >> 2;

adcResult:   1011
4        : & 0100
---------------------
             0000

>> 2     :   0000

----------------------
LED_D6_LAT   :   0000 (OFF)


LED_D7_LAT = (adcResult & 8) >> 3;

adcResult:   1011
8        : & 1000
---------------------
             1000

>> 3     :   0001

----------------------
LED_D7_LAT   :   0001 (ON)
```

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.

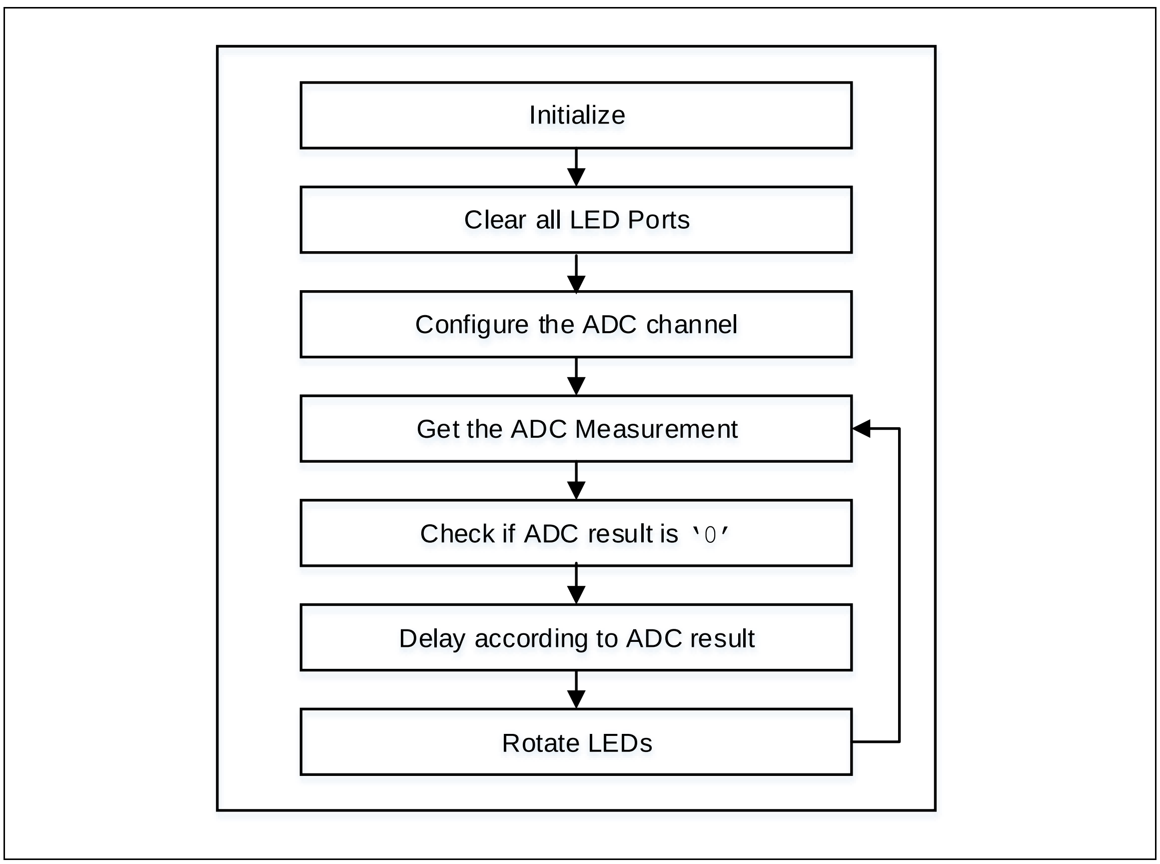
Hardware Effects

Rotate POT1 counterclockwise to see the LEDs shift faster.

Summary

A crucial step in this lesson is to check if the ADC 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/pin_manager.h"
#include "../../mcc_generated_files/adc.h"
#include "../../labHeader.h"

/*
                             Application
 */

#define LAST    16

static uint8_t delay;
static uint8_t adcResult;
static uint8_t rotateReg;

void VSR(void){

    if(labState != RUNNING){
        LED_D4_LAT = LED_D5_LAT = LED_D6_LAT = LED_D7_LAT = OFF;

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

        labState = RUNNING;
    }

    if (labState == RUNNING) {

        delay = adcResult = ADC_GetConversion(POT1) >> 8;

        __delay_ms(5);

        //Delay 2 ms until delay decrements to 0
        while (delay-- != 0){
           __delay_ms(2);
        }

        //Determine which LED will light up
        LED_D4_LAT = rotateReg & 1;
        LED_D5_LAT = (rotateReg & 2) >> 1;
        LED_D6_LAT = (rotateReg & 4) >> 2;
        LED_D7_LAT = (rotateReg & 8) >> 3;

        rotateReg = rotateReg << 1 ;

        //Return to initial position of LED
        if (rotateReg == LAST){
```

```
            rotateReg = 1;
        }
    }

    //Check if a switch event occurs
    if(switchEvent){
        labState = NOT_RUNNING;
    }
}
/**
 End of File
 */
```

```
delay = adcResult = ADC_GetConversion(POT1) >> 8;
```

At `RUNNING` state, the 8 MSbs of the value resulting from the ADC is stored in a static variable '`delay`' which determines the speed of rotation.

```
__delay_ms(5);

//Delay 2 ms until delay decrements to 0
while (delay-- != 0){
     __delay_ms(2);
}
```

A minimum delay of 5 ms is set then the '`delay`' variable decrements until it reaches '`0`'. After which, another delay of 2 ms is set before the code 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 potentiometer POT1 will adjust the brightness of 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, high on time will illuminate the LED brighter. The frequency or period of the PWM does not change. The PWM period is defined as the duration of one cycle or the total amount of on and off time combined. Another important term to take note is the PWM duty cycle which is the ratio of the pulse width to the period and is often expressed in percentage. A lower duty cycle corresponds to less power applied and a higher duty cycle corresponds to more power applied.

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

The PWM period is specified by the PRx register. Timer 2/4/6 is used to count up to the value in CCPRxH combined with two LSBs in CCPxCON. 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

EQUATION 6-1: PWM RESOULUTION

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

Two conditions must hold true for this lesson:

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

Figure 6-1 and **Figure 6-2** show how to configure both the Timer2 and CCP modules for standard PWM operation. Take note that some devices have independent PWM modules instead of a CCP module.

FIGURE 6-1: MCC WINDOW – TMR2 MODULE

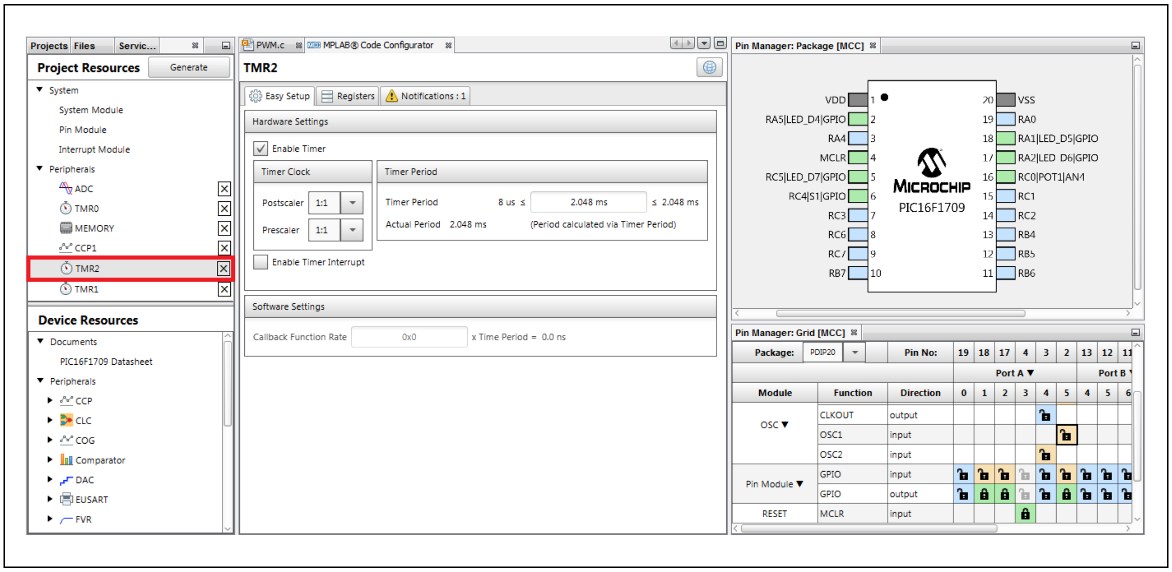
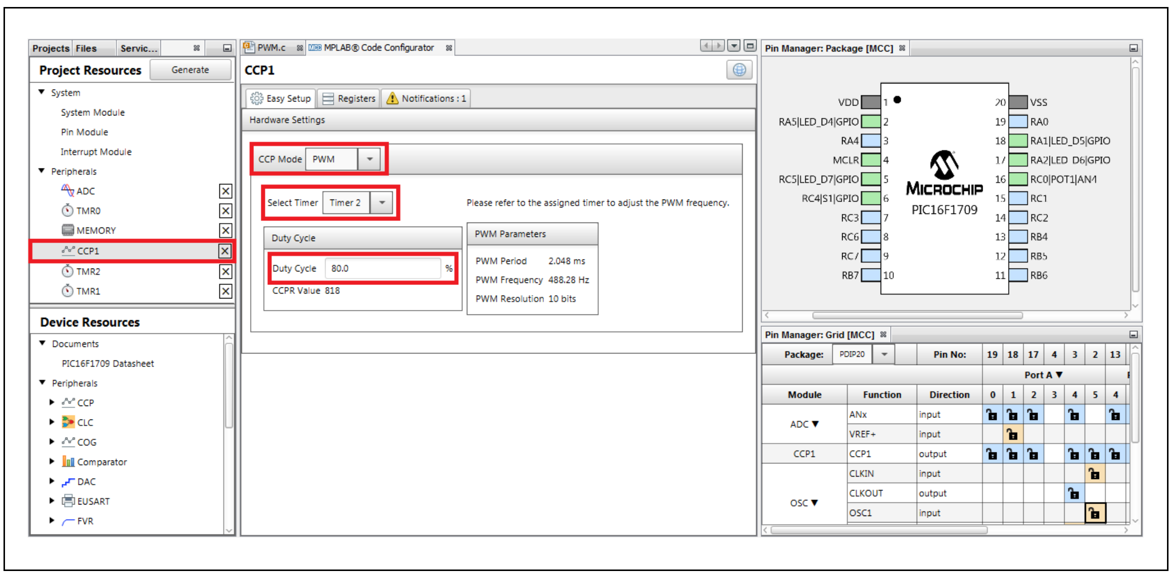


FIGURE 6-2: MCC WINDOW – CCP1::PWM MODULE



C Language

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

EXAMPLE 6.1: 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/pwm1.h"
#include "../../mcc_generated_files/tmr2.h"
#include "../../labHeader.h"

/*
                             Application
 */

uint16_t adcResult;

void PWM(void) {

    if (labState != RUNNING) {
        LED_D4_LAT = LED_D5_LAT = LED_D6_LAT = LED_D7_LAT = OFF;

        //Set RC5 (LED_D7) as output of CCP1 using PPS
        RC5PPS = 0b00001100;

        TMR2_StartTimer();

        labState = RUNNING;
    }

    if (labState == RUNNING) {
        //Start ADC conversion
        adcResult = ADC_GetConversion(POT1) >> 6;

        //Make the adcResult the PWM duty cycle
        PWM1_LoadDutyValue(adcResult);
    }

    //Check if a switch event occurs
    if (switchEvent) {
        TMR2_StopTimer();

        //Restore RC5 (LED_D7) as a normal output
        RC5PPS = 0b00000000;

        labState = NOT_RUNNING;
    }
}
/**
 End of File
 */
```

```
TMR2_StartTimer()
```

This MCC-generated 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 CPP1 module.

```
adcResult = ADC_GetConversion(POT1) >> 6;
```

This statement gets the ADC result from the POT1 channel. Since the ADC module is configured to be left-justified and has a 10-bit resolution, the result is written to the upper 10 bits of the 16-bit return value of `ADC_GetConversion(POT1)`. The result is shifted 6 bits to the right to copy the 10-bit ADC result to the lower 10 bits of the `adcResult` variable.

```
PWM1_LoadDutyValue(adcResult);
```

This uses the `adcResult` as the PWM duty cycle value. This function writes the 8 MSBs and 2 LSBs of the PWM duty cycle to the CPPRL and CCPCON registers, respectively.

```
TMR2_StopTimer();
```

This function stops the Timer2 module by clearing the TMR2ON bit of the T2CON register.

```
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 that 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 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

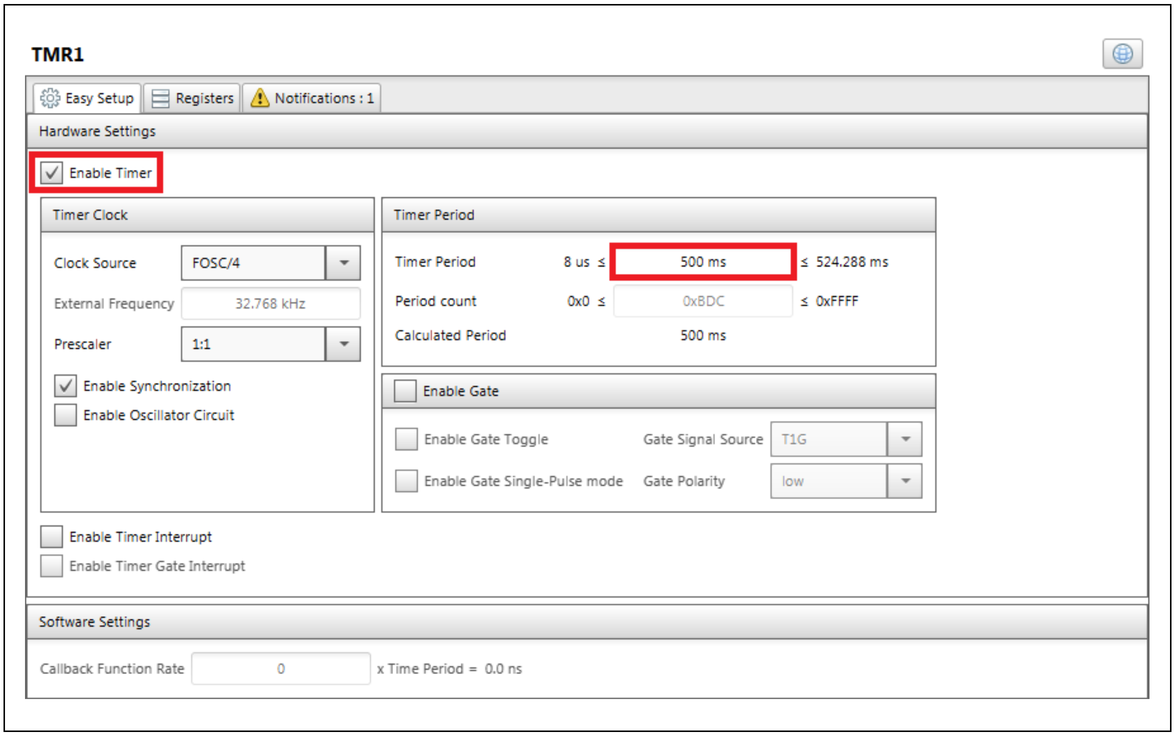
TMR1H and TMR1L are 8-bit registers that form a 16-bit timer/counter register pair. This timer/counter increments from a defined value until it reaches a value of 255 or 0xFF each, and overflows. An overflow will set the Timer1 flag bit ‘high’ and trigger an interrupt when enabled.

PIR1

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

When using MCC, select TMR1 from the list of modules and configure the respective settings as shown in **Figure 7-1**. After generating the source codes, new functions will be made available.

FIGURE 7-1: MCC COMPOSER AREA – TMR1 MODULE



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/pin_manager.h"
#include "../../mcc_generated_files/tmr1.h"
#include "../../labHeader.h"

/*
                              Application
 */

#define LAST   16
```

```
static uint8_t rotateReg;

void Timer1(void) {

    if (labState != RUNNING) {
        LED_D4_LAT = ON;
        LED_D5_LAT = LED_D6_LAT = LED_D7_LAT = OFF;

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

        TMR1_StartTimer();

        labState = RUNNING;
    }

    if (labState == RUNNING) {
        //Wait for Timer1 to overflow
        while (!TMR1IF);

        TMR1_Reload();

        rotateReg = rotateReg << 1;

        //Return to initial position of LED
        if (rotateReg == LAST) {
            rotateReg = 1;
        }

        //Determine which LED will light up
        LED_D4_LAT = rotateReg & 1;
        LED_D5_LAT = (rotateReg & 2) >> 1;
        LED_D6_LAT = (rotateReg & 4) >> 2;
        LED_D7_LAT = (rotateReg & 8) >> 3;

        //Clear the TMR1 interrupt flag
        TMR1IF = 0;
    }

    //Check if a switch event occurs
    if (switchEvent) {
        TMR1_StopTimer();

        labState = NOT_RUNNING;
    }
}
/**
 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 its initial value stated in 'timer1ReloadVal' at 'TMR1_Initialize()' for the delay to be consistent.

During 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 all 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 once the Interrupt Flag is cleared, resumes what it was doing before. It is a rather more efficient way of watching out for events than continuously polling a bit or register.

Hardware Effects

LEDs D4, D5, D6 and D7 rotate from left to right 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 **Lesson 7: Timer1** but this time, we are not continuously watching the TMR1IF flag.

New Register

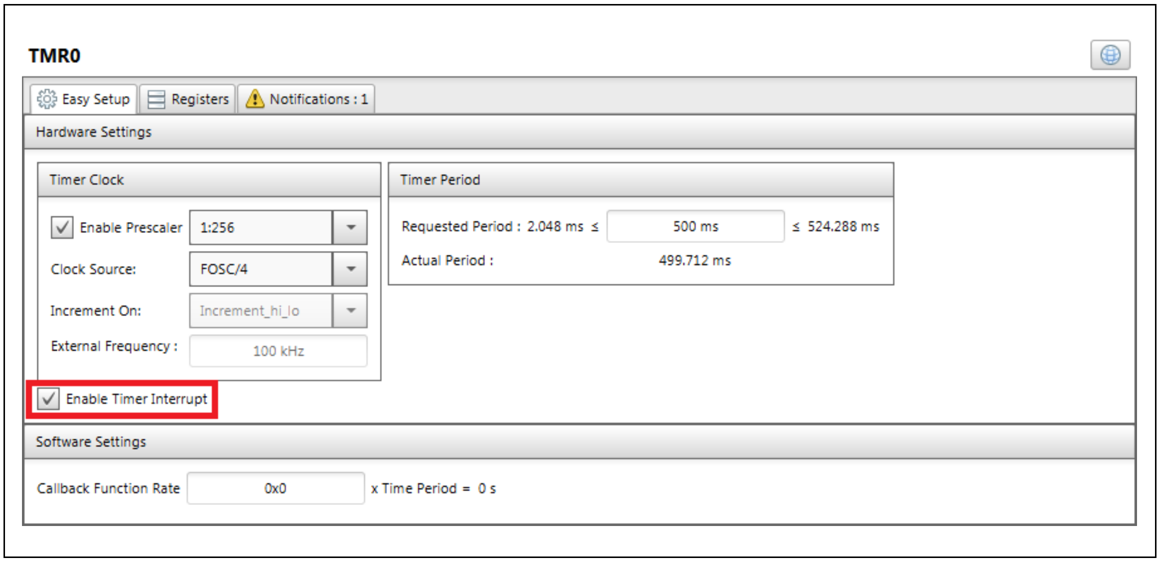
Register	Purpose
INTCON	Contains the various enable and flag bits for the usual interrupt sources.

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

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

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



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

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

#include "../../mcc_generated_files/pin_manager.h"
#include "../../mcc_generated_files/interrupt_manager.h"
#include "../../labHeader.h"

/*
                             Application
 */

void Interrupt(void) {

    if (labState != RUNNING) {
        LED_D4_LAT = ON;
        LED_D5_LAT = LED_D6_LAT = LED_D7_LAT = OFF;

        INTERRUPT_GlobalInterruptEnable();
        INTERRUPT_PeripheralInterruptEnable();

        //Enable the TMR0 Interrupts
        TMR0IE = 1;

        labState = RUNNING;
    }
```

```
    //Check if a switch event occurs
    if (switchEvent) {
        //Disable the TMR0 Interrupts
        TMR0IE = 0;

        INTERRUPT_GlobalInterruptDisable();
        INTERRUPT_PeripheralInterruptDisable();

        labState = NOT_RUNNING;
    }
}
/**
 End of File
 */
```

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. Shown below is the MCC-generated `interrupt_manager.c` code.

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

void interrupt INTERRUPT_InterruptManager (void)
{
   // interrupt handler
    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)
{

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

    TMR0 = timer0ReloadVal;


    // add your TMR0 interrupt custom code
    //If the last LED has been lit, restart the pattern
    if (rotateReg == 1) {
        rotateReg = LAST;
    }

    rotateReg >>= 1;

    //Check which LED should be lit
    LED_D4_LAT = rotateReg & 1;
    LED_D5_LAT = (rotateReg & 2) >> 1;
    LED_D6_LAT = (rotateReg & 4) >> 2;
    LED_D7_LAT = (rotateReg & 8) >> 3;
}
```

```
//Added for Lab Number 8: Interrupts
static uint8_t rotateReg = 1;
```

The static variable `rotateReg` is declared within the `tmr0.c` file. It is used to rotate the light among the LEDs every 500 ms.

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 on RUNNING state, the watchdog timer will start counting. LEDs D4 and D6 are ON while the MCU is in Sleep mode. Pressing the switch won't go to the next lab since the PIC is in Sleep mode. After the watchdog timer has reached its period, which is approximately 4 seconds for this lab, the PIC exits sleep mode and the four LEDs, D4 through D7, are toggled.

Summary

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

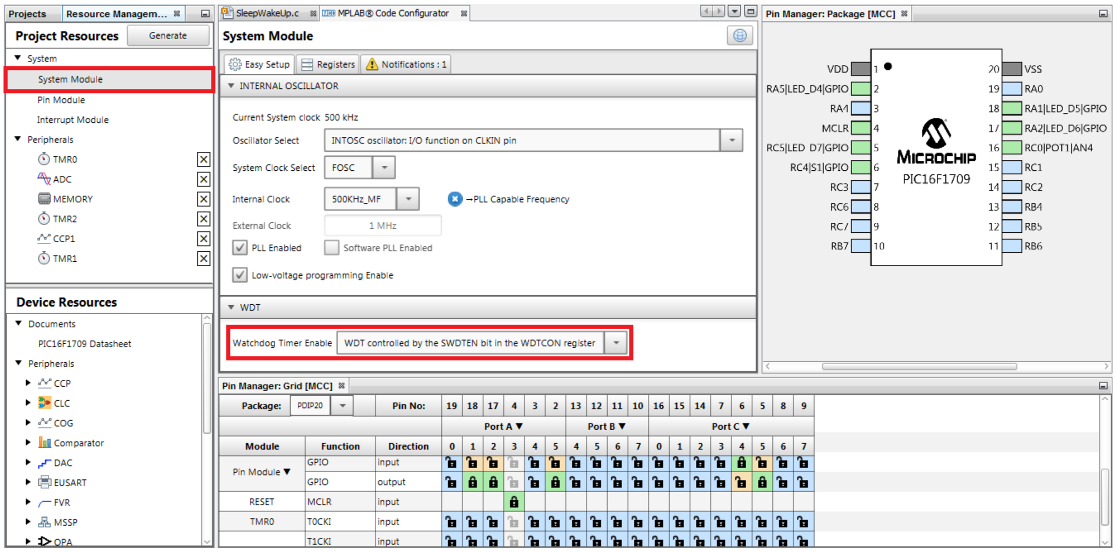
- WDT will be cleared but keeps running, if enabled for operation during Sleep.
- PD bit of the STATUS register is cleared.
- TO bit of the STATUS register is set.
- CPU clock is disabled.

Different PICs have different condition once they enter Sleep mode so it is recommended that the reader refer to the datasheet to know more of these conditions.

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.

WDT can be configured through MCC as shown in **Figure 9-1**.

FIGURE 9-1: MCC WINDOW – WATCHDOG TIMER CONFIGURATION



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

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

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

/*
                             Application
 */

void SleepWakeUp(void) {

    if (labState != RUNNING) {
        LED_D4_LAT = LED_D6_LAT = ON;
        LED_D5_LAT = LED_D7_LAT = OFF;

        //Set the WDT period to 4s
        WDTCONbits.WDTPS = 0b01100;

        //Enable Watchdog Timer
        WDTCONbits.SWDTEN = 1;

        SLEEP();

        labState = RUNNING;
```

```
    }

    if (labState == RUNNING) {
        //Wait 4 seconds for the WDT time out
        //and reverse the states of the LEDs
        LED_D4_LAT = LED_D6_LAT = OFF;
        LED_D5_LAT = LED_D7_LAT = ON;

        //Disable Watchdog Timer
        if (WDTCONbits.SWDTEN) {
            WDTCONbits.SWDTEN = 0;
        }
    }

    //Check if a switch event occurs
    if (switchEvent) {
        labState = NOT_RUNNING;
    }
}
/**
 End of File
 */
```

```
SLEEP();
```

This function tells the PIC to enter 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, which will lose its value when no power is applied. The EEPROM is useful for storing variables that must still be present during no power. It is also convenient to use if the entire RAM space is used up. PIC16F1829 is used for this example and has 256 bytes of EEPROM available. Writes and reads to the EEPROM are relatively quick, and are much faster than program memory operations.

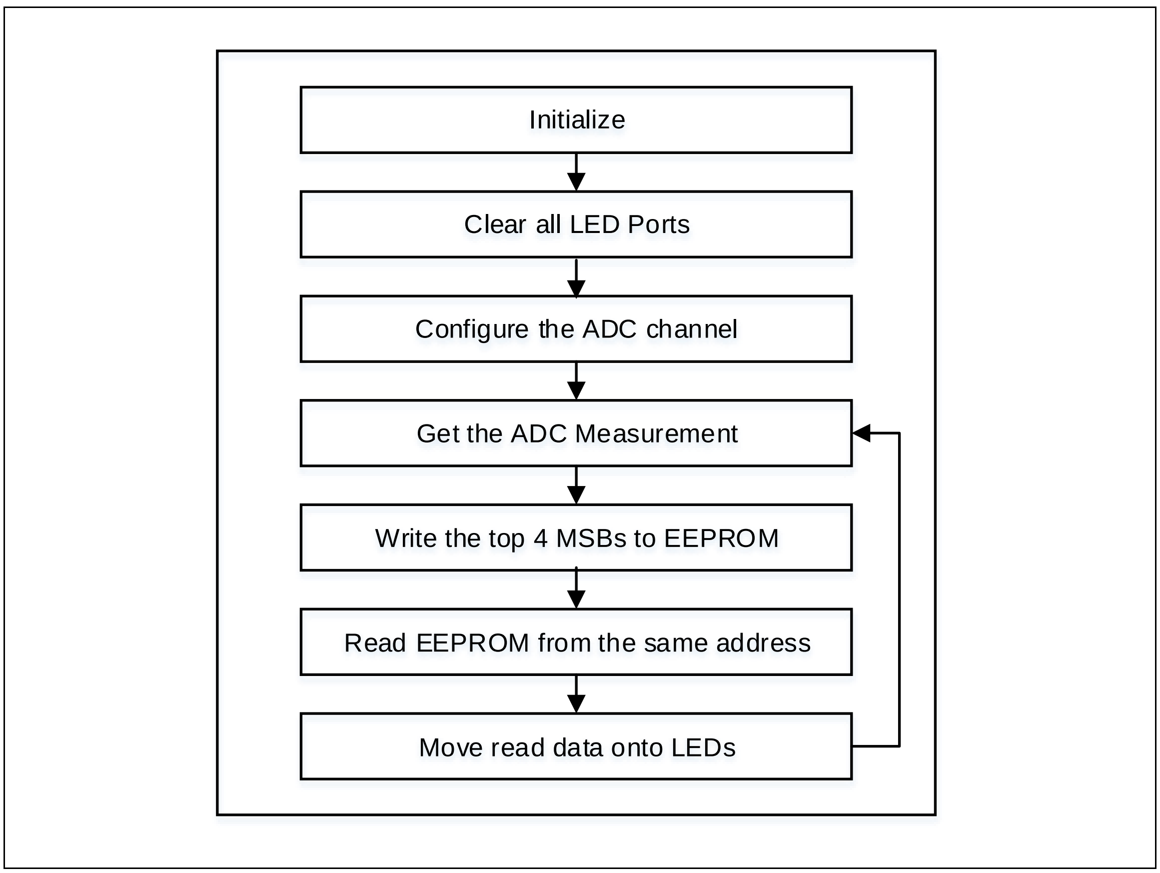
Hardware Effects

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

Summary

This lab has a similar appearance to **LESSON 4: ADC**. But instead of directly moving the ADC result directly onto the LEDs, it performs a simple "write" and "read" on the EEPROM. 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.

FIGURE 10-1: PROGRAM FLOW



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

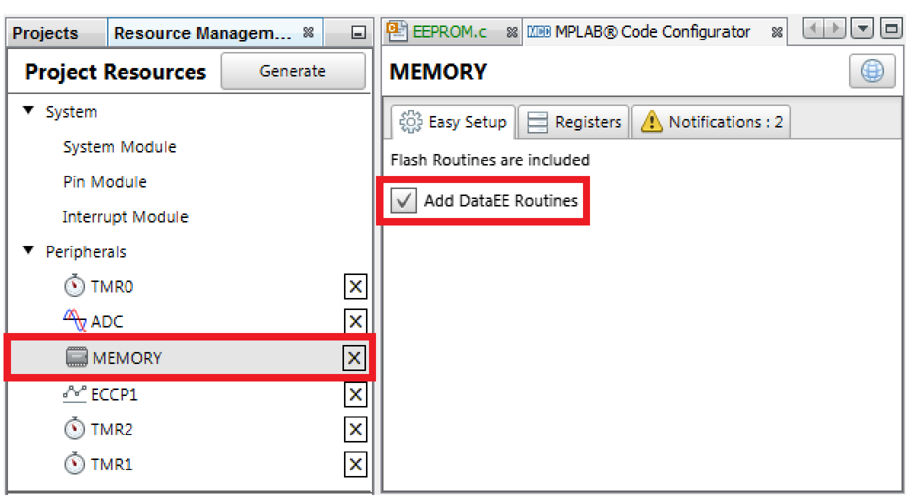
EEADRH:EEADRL form a register pair which holds the 15-bit address of the program memory location being read.

MCC Instructions

Instruction	Purpose
`DATAEE_WriteByte(uint8_t bAdd, uint8_t bData)`	Writes a data byte `bData` to Data EEPROM address `bAdd`
`DATAEE_ReadByte(uint8_t bAdd)`	Reads a data byte from Data EEPROM address `bAdd`

The instructions above are automatically generated by the MCC when the Memory module is configured as shown in **FIGURE 10-2**.These functions can be found in `memory.c`.

FIGURE 10-2: MCC WINDOW – MEMORY MODULE TO GENERATE DATAEE ROUTINES



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/pin_manager.h"
#include "../../mcc_generated_files/adc.h"
#include "../../mcc_generated_files/memory.h"
#include "../../labHeader.h"

/*
                             Application
 */

static uint8_t adcResult;
static uint8_t ledDisplay;

void EEPROM(void) {

    if (labState != RUNNING) {
        LED_D4_LAT = LED_D5_LAT = LED_D6_LAT = LED_D7_LAT = OFF;

        labState = RUNNING;
    }

    if (labState == RUNNING) {
        uint8_t eeAddr = 0x00;

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

        //Load whatever is in EEPROM to the LED Display
        ledDisplay = DATAEE_ReadByte(eeAddr);

        //Determine which LEDs will light up
        LED_D4_LAT = ledDisplay & 1;
        LED_D5_LAT = (ledDisplay & 2) >> 1;
        LED_D6_LAT = (ledDisplay & 4) >> 2;
        LED_D7_LAT = (ledDisplay & 8) >> 3;
    }

    //Check if a switch event occurs
    if (switchEvent) {
        labState = NOT_RUNNING;
    }
}
/*
 End of File
 */
```

```
uint8_t eeAddr = 0x00;
```

For this lab, we are going to access EEPROM address ‘0x00’. Please see your device datasheet for valid EEPROM address range.

```
DATAEE_WriteByte(eeAddr, adcResult);
```

This function writes the values stored within ‘adcResult’ (see **LESSON 4: ADC**) to the data EEPROM memory at address eeAddr.

```
ledDisplay = DATAEE_ReadByte(eeAddr);
```

The function above reads the EEPROM data byte located at address eeAddr then stores the read data to a user-defined global variable ‘ledDisplay’. This data will be reflected on the LED ports.

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

LEDs 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.

MCC Instructions

Instruction	Purpose
`FLASH_WriteWord(uint16_t flashAddr, uint16_t *ramBuf, uint16_t word)`	Writes the given word on the given flash address `flashAddr`
`FLASH_ReadWord(uint16_t flashAddr)`	Reads a word from the given flash address `flashAddr`

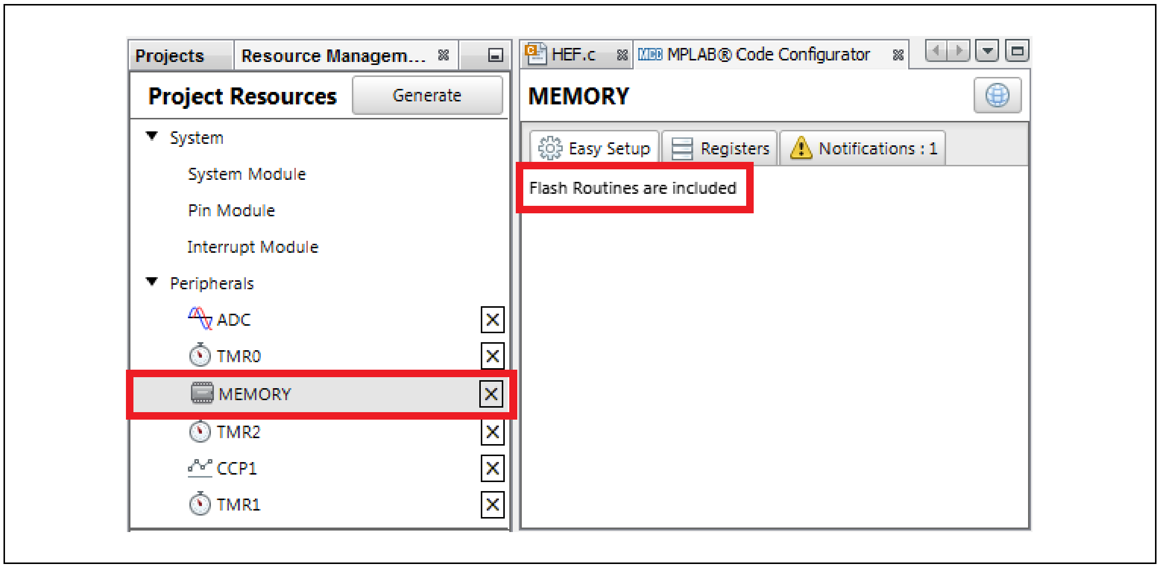
The instructions above are automatically generated by the MCC when the Memory module is selected (see **Figure 11-1**). These functions can be found in `memory.c`.

Declaration	Purpose
`WRITE_FLASH_BLOCKSIZE`	Maximum number of words that can be written in one block write.
`ERASE_FLASH_BLOCKSIZE`	Number of words in one erase block.

The macro declarations above can be found on the MCC-generated `memory.h`.

A block is the minimum program flash memory size that can be erased by user software. Before writing to a program memory, the block where the word(s) should be written to must be erased. Please see your device datasheet for valid HEF memory address range.

FIGURE 11-1: MCC WINDOW – MEMORY MODULE TO GENERATE FLASH ROUTINES



C Language

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

EXAMPLE 11.1: C CODE FOR "HEF" LAB

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

#include "../../mcc_generated_files/pin_manager.h"
#include "../../mcc_generated_files/memory.h"
#include "../../labHeader.h"

/*
                             Application
 */

static uint8_t rotateReg;

void HEF(void) {

    if (labState != RUNNING) {
        LED_D4_LAT = LED_D5_LAT = LED_D6_LAT = LED_D7_LAT = OFF;

        labState = RUNNING;
    }

    if (labState == RUNNING) {
        uint16_t writeData = 0x0005;
        uint16_t HefAddr = 0x1F80;
        uint16_t Buf[ERASE_FLASH_BLOCKSIZE];

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

```
        //Read back value and store to LED display
        rotateReg = FLASH_ReadWord(HefAddr);

        //Determine which LED will light up
        //ie. which bit in the register the 1 has rotated to.
        LED_D4_LAT = rotateReg & 1;
        LED_D5_LAT = (rotateReg & 2) >> 1;
        LED_D6_LAT = (rotateReg & 4) >> 2;
        LED_D7_LAT = (rotateReg & 8) >> 3;
    }

    //Check if a switch event occurs
    if (switchEvent) {
        labState = NOT_RUNNING;
    }
}
/**
 End of File
 */
```

```
uint16_t writeData = 0x0005;
```

Data to be written is equal to '5'.

```
uint16_t HefAddr = 0x1F80;
```

HEF memory address `0x1F80` is selected for this lab. For this example, the HEF memory address range is 1F80h to 1FFFh (using PIC16F1709).

```
uint16_t Buf[ERASE_FLASH_BLOCKSIZE];
```

This is a declaration for an array with size `ERASE_FLASH_BLOCKSIZE`.

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

This routine saves all existing data within the block where `HefAddr` is located to the previously declared array `Buf`. Thus, it is necessary for `Buf` to be declared with size of at least one erase block. The location where the new data (`writeData`) will be written to within the buffer is automatically identified by software. The updated data words in `Buf` are then written to one complete block in the HEF memory.

```
rotateReg = FLASH_ReadWord(HefAddr);
```

This function reads the data from the previously written HEF memory address `HefAddr` and reflects them on the LEDs.