TB3270

Getting Started with PWM Using CCP on PIC18

Introduction

Authors: Grig Barbulescu, Iustinian Bujor, Microchip Technology Inc.

This technical brief provides information about Capture/Compare/PWM (CCP) and Pulse-Width Modulation (PWM) peripherals and intends to familiarize the user with PIC® microcontrollers. The CCP is a peripheral that times and controls different events and generates Pulse-Width Modulation (PWM) signals. The PWM peripheral generates a pulse-width modulated signal determined by the duty cycle and period and that can be configured by the user.

This document describes the application area and the modes of operation, as well as the hardware and software requirements of the PWM module and of the CCP module configured in PWM mode.

Throughout the document, the configuration of the peripherals will be described in detail. The descriptions will start with the input Timer selection, the mode of operation and how to change the period, in addition to the duty cycle and resolution to generate the desired PWM signal.

This document covers the following use cases:

- **Configuring a PWM Signal Frequency and Duty Cycle:**
  How to configure the CCP in conjunction with a Timer to generate a low-speed PWM signal with configurable frequency and duty-cycle.
- **Generating a PWM Signal with Constant On-Time and Variable Frequency:**
  How to configure the CCP in conjunction with a Timer to generate a PWM signal with a constant on-time of one microsecond and variable frequency.
- **RGB-LED Dimming Using PWM:**
  How to configure the CCP and PWM in conjunction with a Timer to generate three PWM signals that will produce a color game on an RGB LED, based on the dimming effect.

For each use case, there are two different software implementations that have the same functionalities: one code generated with MPLAB® Code Configurator (MCC) and one bare metal code.

The MCC generated code offers hardware abstraction layers that make using code across different devices from the same family easier. Additionally, the bare metal code is easier to follow and allows for a fast ramp-up on the use case associated code.

Note: The examples in this technical brief have been developed using PIC18F47Q10 Curiosity Nano development board. The PIC18F47Q10 pin package present on the board is QFN40.

Table of Contents

1. Peripheral Overview

The CCP peripheral can operate in one of the three modes:

- Capture
- Compare
- Pulse-Width Modulation (PWM)

This technical brief focuses on presenting the PWM mode of the CCP peripheral. In PWM mode, it can generate pulse-width modulated signals with configurable frequency and duty cycle.

The PWM peripheral is a simplified version of the CCP module and designed to function only in PWM mode.

The PWM provides power to a load by quickly switching between two states: ON and OFF. The PWM signal consists of a square wave where the high portion, also known as pulse-width, of the signal is considered the ON state while the low portion of the signal is considered the OFF state. The three main characteristics of a PWM signal are: period, duty cycle and resolution.

The PWM period is defined as the duration of one complete cycle or the total amount of On and Off time combined.

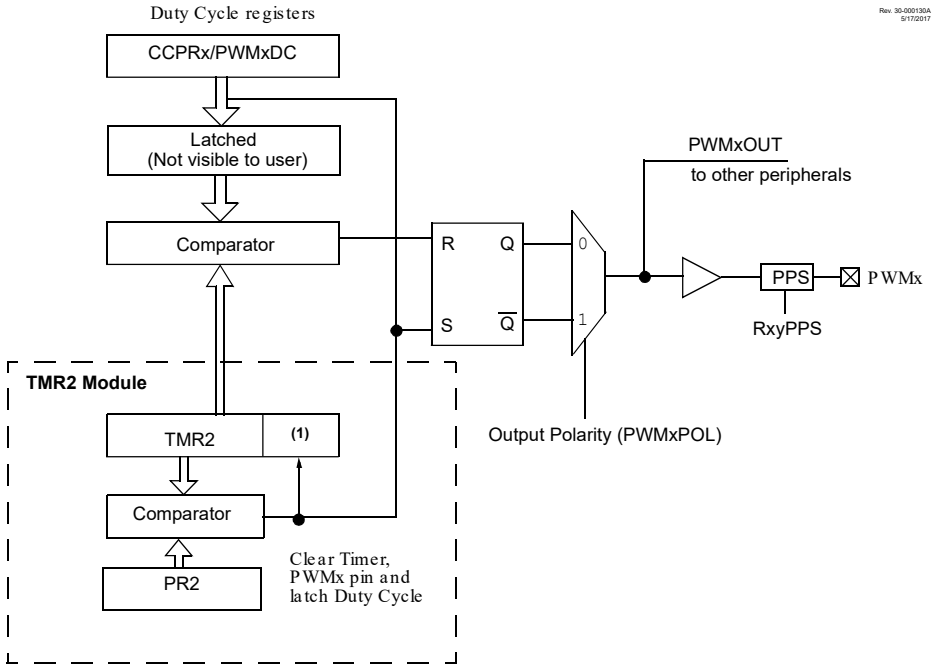
The PWM duty cycle describes the proportion of the On time to the Off time and it is expressed in percentages where 0% is fully Off and 100% is fully On. A lower duty cycle corresponds to less power applied and a higher duty cycle corresponds to more power applied.

The pulse-width can vary in time and is defined in steps. The PWM resolution defines the maximum number of steps that can be present in a single PWM period. A higher resolution allows for more precise control of the pulse-width time, which then influences the power that is applied to the load.

For the CCP to operate in PWM mode, Timer2/4/6 must be linked to it. The period of the PWM signal generated by CCP in PWM mode is represented by the period register of the respective Timer. It is required for the Timer to have $F_{OSC}/4$ as clock input for correct PWM operation.

The following figure shows a simplified block diagram of the CCP module in PWM mode of operation.

Figure 1-1. PWM Block Diagram



Note 1: 8-bit timer is concatenated with the two Least Significant bits of 1/FOSC adjusted by the Timer2 prescaler to create a 10-bit time base.

2. Configuring a PWM Signal Frequency and Duty Cycle

This example shows how to initialize the CCP1 peripheral in PWM mode, the Timer2 and other software and hardware requirements to generate a low-speed PWM signal with configurable frequency and duty cycle.

The configuration of the PWM parameters is done at run time and through the usage of a button. This allows the configuration to differentiate long presses from short presses.

When a short press occurs, the value of the PWM duty cycle increases in steps of 25%. Its minimum value is 25% and its maximum value is 75%. Whenever the maximum value is reached, a new short press will cause a transition to the minimum value.

When a long press occurs, the value of the PWM frequency increases in steps of one Hz. Its minimum value is 1 Hz and its maximum value is 4 Hz. Whenever the maximum value is reached, a new long press will cause a transition to the minimum value.

To achieve the functionality described by the use case, the following actions will have to be performed:

- System clock initialization
- Port initialization
- Timer2 initialization
- CCP1 initialization
- PPS initialization

2.1 MCC Generated Code

To generate this project using MPLAB® Code Configurator (MCC), follow the next steps:

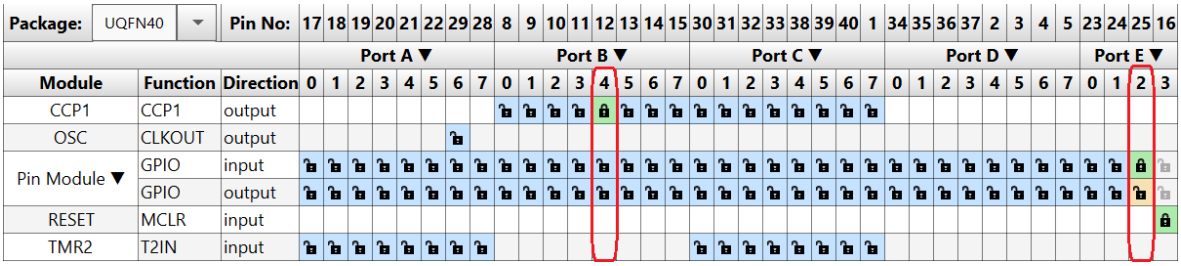
1. Create a new MPLAB X IDE project for PIC18F47Q10.
2. Open the MCC from the toolbar. Information about how to install the MCC plug-in can be found here.
3. Go to *Project Resources → System → System Module* and do the following configuration:
   - Oscillator Select: LFINTOSC
   - Clock Divider: 1
   - In the Watchdog Timer Enable field from the **WWDT** tab, make sure **WDT Disabled** is selected.
   - In the **Programming** tab, make sure **Low-Voltage Programming Enable** is checked.
4. From the Device Resources window, add TMR2 and CCP1. Do the following configurations for each peripheral:
   Timer2 Configuration:
   - **Hardware Settings** tab
     - Enable Timer: checked
     - Control Mode: Roll over pulse
     - Start/Reset Option: Software control
   - **Timer Clock** tab
     - Clock Source: $F_{OSC}/4$
     - Prescaler: 1:32
     - Postscaler: 1:1
   - **Timer Period** tab
     - Timer Period: 1.057s
   - **Software Settings** tab
     - Enable Timer Interrupt: unchecked

   CCP1 Configuration:
   - Enable CCP: checked
   - CCP Mode: PWM
   - Select Timer: Timer2

  - Duty Cycle: 25%
  - CCPR Alignment: right_aligned

5. Open *Pin Manager → Grid View* window, select **UQFN40** in the Package field and do the following pin configurations:
   - Set Port B pin 4 (RB4) as output for CCP1
   - Set Port E pin 2 (RE2) as GPIO input

Figure 2-1. Pin Mapping



6. Go to *Project Resources → System → Pin Module → Easy Setup* and enable WPU for the RE2 pin.
7. Click **Generate** in the **Project Resources** tab.
8. In the `main.c` file generated by MCC, add the following code:
   - `ButtonCheck()` function
   - Button press handling

   The `ButtonCheck()` function is used to detect if the button is pressed and, if so, to differentiate a long button press from a short button press. The method used is the incrementation of a counter. As long as the button is pressed, the counter value increments continuously and, after that, a check is performed to determine if the counter value is smaller or greater than a specific threshold. The function can return one of the three states: `BT_NOCHANGE`, `BT_SHORT_PRESS`, `BT_LONG_PRESS`.

   The button press handling is done inside the `while(1)` loop. The `ButtonCheck()` function is called continuously and, depending on the value that it returned, the PWM frequency and duty cycle change, as referenced earlier in this use case.

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

2.2 Bare Metal Code

The necessary code and functions to implement the presented example are analyzed in this section.

The first step is to configure the microcontroller to disable the Watchdog Timer and to enable Low-Voltage Programming.

```
#pragma config WDTE = OFF
#pragma config LVP = ON
```

The internal oscillator must be set to the desired value. Since the purpose of this example is to generate a low-speed PWM signal, the Low-Frequency Oscillator was used. The value of the clock frequency will be 31 kHz. This translates into the following function:

```
static void CLK_Initialize(void)
{
    /* Configure NOSC LFINTOSC; NDIV 1 FOSC = 31kHz */
    OSCCON1bits.NOSC = 5;
    OSCCON1bits.NDIV = 0;
}
```

The output driver for I/O pin RB4 must be enabled since it will be used as CCP1 output while RE2 must be configured as analog input with internal pull-up enabled. This translates into the following function:

```
static void PORT_Initialize(void)
{
    /* RB4 is output for PWM1 */
    TRISBbits.TRISB4 = 0;
    /* RE2 is digital input with pull-up for push-button */
    TRISEbits.TRISE2 = 1;
    ANSELEbits.ANSELE2 = 0;
    WPUEbits.WPUE2 = 1;
}
```

For Timer2 to use as clock source $F_{OSC}/4$ with a 1:32 clock prescaler, the following function is used:

```
#define _XTAL_FREQ                      31000UL

#define FREQUENCY_MIN                   1    /* Hz */
#define TIMER_PRESCALER                 32   /* 1:32 */
#define FREQUENCY_TO_PR_CONVERT(F)      (uint8_t)(((_XTAL_FREQ)/ \
                                        (4*(F))/(TIMER_PRESCALER))-1)
static void TMR2_Initialize(void)
{
    /* TIMER2 clock source is FOSC/4 */
    T2CLKCONbits.CS = 1;

    /* TIMER2 counter reset */
    T2TMR = 0x00;

    /* TIMER2 ON, prescaler 1:32, postscaler 1:1 */
    T2CONbits.OUTPS = 0;
    T2CONbits.CKPS = 5;
    T2CONbits.ON = 1;

    /* Configure the default period */
    PR2 = FREQUENCY_TO_PR_CONVERT(FREQUENCY_MIN);
}
```

CCP1 is configured in PWM mode and uses Timer2 as period generator. The value from the CCPR1 register is left-aligned and the initial duty cycle value is set at 25%. This translates into the following function:

```
#define _XTAL_FREQ                       31000UL

#define FREQUENCY_MIN                    1   /* Hz */
#define DUTYCYCLE_MIN                    25  /* percents */

#define DUTYCYCLE_TO_CCPR_CONVERT(D,F)   (uint16_t)((float)(D)*(((_XTAL_FREQ)/ \
                                         (F))/(TIMER_PRESCALER))-1)/100.0)

static void PWM1_Initialize(void)
{
    /* MODE PWM; EN enabled; FMT left_aligned */
    CCP1CONbits.MODE = 0x0C;
    CCP1CONbits.FMT = 1;
    CCP1CONbits.EN = 1;

    /* Selecting Timer 2 */
    CCPTMRSbits.C1TSEL = 1;

    /* Configure the default duty cycle */
    CCPR1 = (DUTYCYCLE_TO_CCPR_CONVERT(DUTYCYCLE_MIN, FREQUENCY_MIN)) << 6;
}
```

Configuring the location of the pins is independent of the application purpose and the CCP mode. Each microcontroller has its own default physical pin position for peripherals, though the pin positions can be changed using the Peripheral Pin Select (PPS).

Therefore, the CCP pins can be relocated using the CCP1PPS and CCP2PPS registers for the input channels while using the RxyPPS registers for the output channels. The PPS configuration values can be found in the *Peripheral Pin*

Select Module section of a device data sheet. For this example, the output PWM signal from CCP1 will be routed to the RB4 pin. This translates into the following code:

```
static void PPS_Initialize(void)
{
    /* Configure RB4 for PWM1 output */
    RB4PPS = 0x05;
}
```

At run time, the PWM generator is configured using these functions for duty cycle and period values respectively:

```
static   void PWM1_LoadDutyValue(uint16_t dutyValue)
{
    /* Only the upper 10 bits are used in register CCPR1 */
    CCPR1 = dutyValue << 6;
}

static  void TMR2_LoadPeriodRegister(uint8_t periodVal)
{
    /* Configure the period register */
    PR2 = periodVal;
}
```

To achieve the functionality presented in the description of this use case, the following functions need to be implemented:

- `ButtonCheck()` function
- Button press handling

The `ButtonCheck()` function is used to detect if the button is pressed and, if so, to differentiate a long button press from a short button press. The method used is the incrementation of a counter. As long as the button is pressed, the counter value increments continuously and, after that, a check is performed to determine if the counter value is smaller or greater than a specific threshold. The function can return one of the three states: `BT_NOCHANGE`, `BT_SHORT_PRESS`, `BT_LONG_PRESS`.

The button press handling is done inside the `while(1)` loop. The `ButtonCheck()` function is called continuously and, depending on the value that it returned, the PWM frequency and duty cycle changes, as referenced earlier in this use case.

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

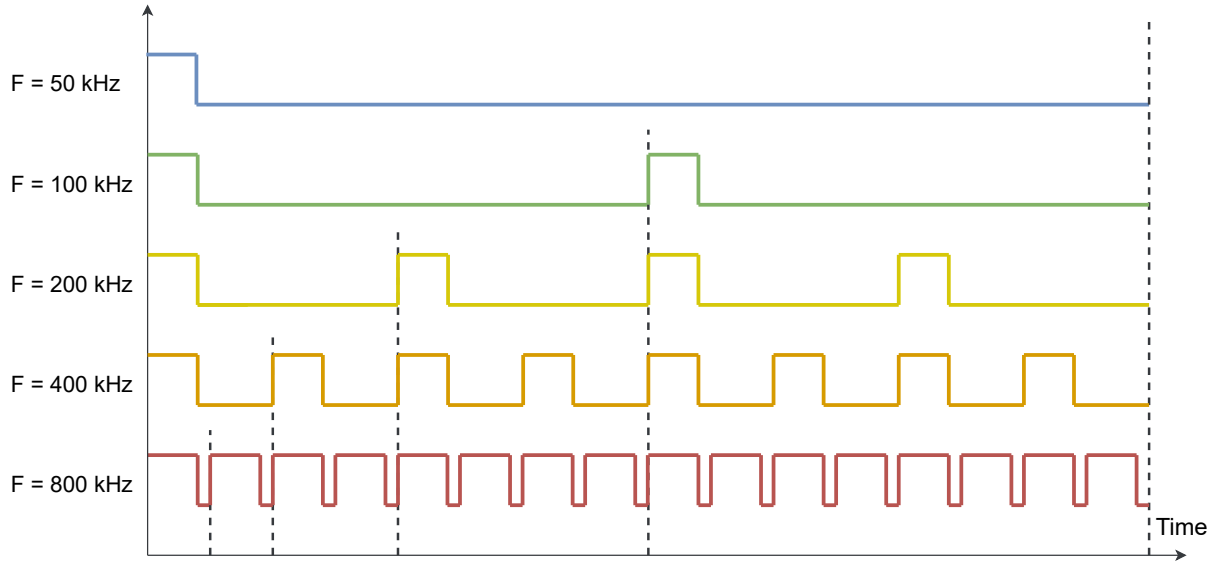
3. Generating a PWM Signal with Constant On-Time and Variable Frequency

This example shows the initialization of the CCP2 peripheral in PWM mode, the initialization of Timer4 and other software and hardware requirements to generate a PWM signal with a constant pulse-width of one microsecond and variable frequency.

The configuration of the PWM frequency is done through the usage of a button.

When a press occurs, the PWM frequency value doubles. Its minimum value is 50 kHz and its maximum value is 800 kHz. Whenever the maximum value is reached, a new press will cause a transition to the minimum value.

Figure 3-1. PWM Signals for Each Frequency



To achieve the functionality described by the use case, the following actions must be performed:

- System clock initialization
- Port initialization
- Timer4 initialization
- CCP2 initialization
- PPS initialization

3.1 MCC Generated Code

To generate this project using MPLAB® Code Configurator (MCC), follow the next steps:

1. Create a new MPLAB X IDE project for PIC18F47Q10.
2. Open the MCC from the toolbar. Information about how to install the MCC plug-in can be found here.
3. Go to *Project Resources → System → System Module* and do the following configuration:
   - Oscillator Select: HFINTOSC
   - HF Internal Clock: 64_MHz
   - Clock Divider: 1
   - In the Watchdog Timer Enable field in the **WWDT** tab, make sure **WDT Disabled** is selected.
   - In the **Programming** tab, make sure **Low-Voltage Programming Enable** is checked.
4. From the Device Resources window, add TMR4 and CCP2. Do the following configurations for each peripheral:
   Timer4 Configuration:
   - **Hardware Settings** tab

- Enable Timer: checked
- Control Mode: Roll over pulse
- Start/Reset Option: Software control

– **Timer Clock** tab
- Clock Source: $F_{OSC}/4$
- Prescaler: 1:2
- Postscaler: 1:1

– **Timer Period** tab
- Timer Period: 20 us

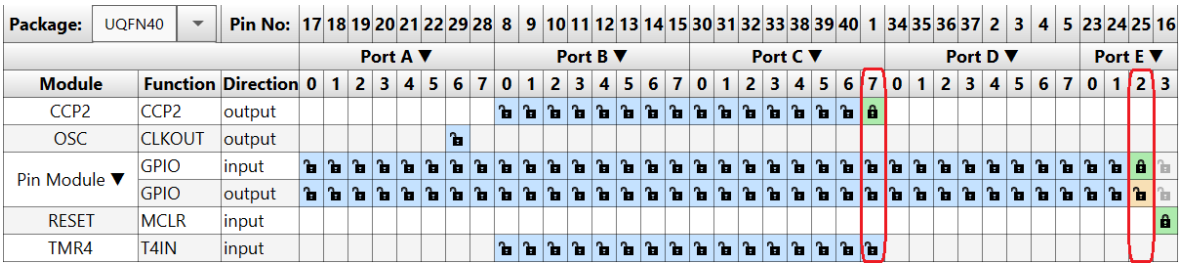
– **Software Settings** tab
- Enable Timer Interrupt: unchecked

CCP2 Configuration:

– Enable CCP: checked
– CCP Mode: PWM
– Select Timer: Timer4
– Duty Cycle: 5%
– CCPR Alignment: right_aligned

5. Open *Pin Manager → Grid View* window, select **UQFN40** in the Package field and do the following pin configurations:
   – Set Port C pin 7 (RC7) as output for CCP2
   – Set Port E pin 2 (RE2) as GPIO input

Figure 3-2. Pin Mapping



6. Go to *Project Resources → System → Pin Module → Easy Setup* and enable WPU for the RE2 pin.
7. Click **Generate** in the **Project Resources** tab.
8. In the `main.c` file generated by MCC, add the following code:
   – `ButtonCheck()` function
   – Button press handling

The `ButtonCheck()` function is used to detect if the button is pressed. Whenever a press is detected, a 10 ms delay is applied to avoid the bouncing effect and the program waits until the button is released. The function can return one of the two states: `BT_NOCHANGE` or `BT_PRESS`.

```
void main(void)
{
    uint8_t index = 0;
    /*  Initialize the device */
    SYSTEM_Initialize();

    while (1)
    {
        if(ButtonCheck() == BT_PRESS)
        {
            /* When a button press is detected, the index is updated */
            index++;
            if(index >= FREQUENCY_LIST_DIMENSION)
                index = 0;
```

```
            /* and the frequency is changed to the next one in the list */
            TMR4_LoadPeriodRegister(frequencies_list[index]);
        }
    }
}
```

The `frequencies_list` array contains the selectable PWM frequencies presented in the description of this use case (50 kHz, 100 kHz, 200 kHz, 400 kHz, and 800 kHz).

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

3.2 Bare Metal Code

The necessary code and functions to implement the presented example are analyzed in this section.

The first step is to configure the microcontroller to disable the Watchdog Timer and to enable Low-Voltage Programming.

```
#pragma config WDTE = OFF
#pragma config LVP = ON
```

The internal oscillator must be set to the desired value. This example uses the HFINTOSC with a frequency of 64 MHz. This translates into the following function:

```
static void CLK_Initialize(void)
{
    /* Configure NOSC HFINTOSC; NDIV 1; FOSC = 64MHz */
    OSCCON1bits.NOSC = 6;
    OSCCON1bits.NDIV = 0;

    /* HFFRQ 64_MHz */
    OSCFRQbits.HFFRQ = 8;
}
```

The output driver for I/O pin RC7 must be enabled since it will be used as CCP2 output. RE2 must be configured as analog input with internal pull-up enabled. This translates into the following function:

```
static void PORT_Initialize(void)
{
    /* RC7 is output for PWM2 */
    TRISCbits.TRISC7 = 0;
    /* RE2 is digital input with pull-up for push-button */
    TRISEbits.TRISE2 = 1;
    ANSELEbits.ANSELE2 = 0;
    WPUEbits.WPUE2 = 1;
}
```

For Timer4 to use as clock source $F_{OSC}/4$ with a 1:2 clock prescaler, the following function is used:

```
static void TMR4_Initialize(void)
{
    /* TIMER4 clock source is FOSC/4 */
    T4CLKCONbits.CS = 1;

    /* TIMER4 counter reset */
    T4TMR = 0x00;

    /* TIMER4 ON, prescaler 1:2, postscaler 1:1 */
    T4CONbits.OUTPS = 0;
    T4CONbits.CKPS = 1;
    T4CONbits.ON = 1;

    /* Configure initial period register value */
```

```
    PR4 = frequencies_list[0];
}
```

CCP2 is configured in PWM mode with a constant pulse-width of one microsecond and uses Timer4 as period generator. This translates into the following function:

```
static void PWM2_Initialize(void)
{
    /* MODE PWM; EN enabled; FMT right_aligned */
    CCP2CONbits.MODE = 0x0C;
    CCP2CONbits.FMT = 0;
    CCP2CONbits.EN = 1;

    /* Constant on-time setting is 1 us */
    CCPR2 = 32;

    /* Select timer 4 */
    CCPTMRSbits.C2TSEL = 2;
}
```

Configuring the location of the pins is independent of the application purpose and the CCP mode. Each microcontroller has its own default physical pin position for peripherals, but the pin positions can be changed using the Peripheral Pin Select (PPS).

Therefore, the CCP pins can be relocated using the CCP1PPS and CCP2PPS registers for the input channels while using the RxyPPS registers for the output channels.

The PPS configuration values can be found in the *Peripheral Pin Select Module* section of a device data sheet. For this example, the output PWM signal from CCP2 will be routed to RC7 pin. This translates into the following code:

```
static void PPS_Initialize(void)
{
    /* Configure RC7 for PWM2 output */
    RC7PPS = 0x06;
}
```

To achieve the functionality presented in the description of this use case, the following functions need to be implemented:

- `ButtonCheck()` function
- Button press handling
- `TMR4_loadPeriodRegister()` function

The `ButtonCheck()` function is used to detect if the button is pressed. Whenever a press is detected, a delay of 10 ms is applied to avoid the bouncing effect and the program waits until the button is released. The function can return one of the two states: `BT_NOCHANGE` or `BT_PRESS`.

```
static void TMR4_LoadPeriodRegister(uint8_t periodVal)
{
    /* Configure the period register */
    PR4 = periodVal;
}

void main(void)
{
    uint8_t index = 0;
    /*  Initialize the device */
    PORT_Initialize();
    PPS_Initialize();
    CLK_Initialize();
    TMR4_Initialize();
    PWM2_Initialize();

    while (1)
    {
        if(ButtonCheck() == BT_PRESS)
        {
            /* When a button press is detected, the index is updated */
            index++;
            if(index >= FREQUENCY_LIST_DIMENSION)
```

```
                index = 0;

            /* and the frequency is changed to the next one in the list */
            TMR4_LoadPeriodRegister(frequencies_list[index]);
        }
    }
}
```

The `frequencies_list` array contains the selectable PWM frequencies presented in the description of this use case (50 kHz, 100 kHz, 200 kHz, 400 kHz and 800 kHz).

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

4. RGB-LED Dimming Using PWM

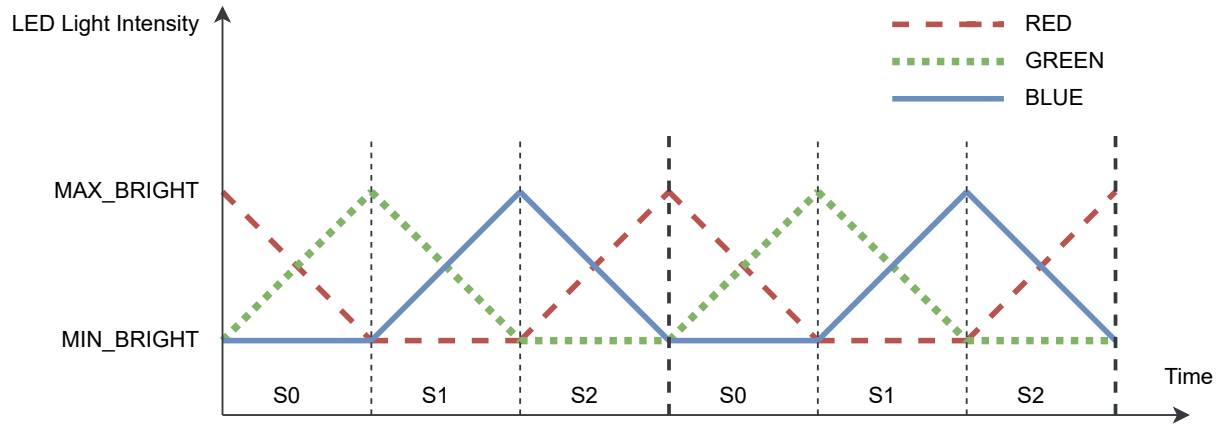
This example shows the initialization of the CCP1, CCP2, PWM3, Timer2 peripherals as well as other software and hardware requirements to generate three PWM signals. This signal generation will then produce a color game on an RGB LED, based on the dimming effect.

The color game is based on the dimming effect applied on an RGB LED. It is composed of three steps:

- The red channel decreases its brightness from fully On to fully Off as the green channel increases its brightness from fully Off to fully On. During this step, the blue channel is turned fully Off.
- The green channel decreases its brightness from fully On to fully Off as the blue channel increases its brightness from fully Off to fully On. During this step, the red channel is turned fully Off.
- The blue channel decreases its brightness from fully On to fully Off as the red channel increases its brightness from fully Off to fully On. During this step, the green channel is turned fully Off.

The three steps are repeating circularly and the updates of the duty cycles are triggered by interrupts.

Figure 4-1. LED Light Intensity for Each Channel



To achieve the functionality described by the use case, the following actions must be performed:

- System clock initialization
- Port initialization
- Timer2 initialization
- CCP1 initialization
- CCP2 initialization
- PWM3 initialization
- PPS initialization
- Interrupts initialization
- Timer2 interrupt handling

4.1 MCC Generated Code

To generate this project using MPLAB® Code Configurator (MCC), follow the next steps:

1. Create a new MPLAB X IDE project for PIC18F47Q10.
2. Open the MCC from the toolbar. Information about how to install the MCC plug-in can be found here.
3. Go to *Project Resources → System → System Module* and do the following configuration:
   - Oscillator Select: HFINTOSC
   - HF Internal Clock: 64_MHz
   - Clock Divider: 1
   - In the Watchdog Timer Enable field in the **WWDT** tab, make sure **WDT Disabled** is selected.
   - In the **Programming** tab, make sure **Low-Voltage Programming Enable** is checked.

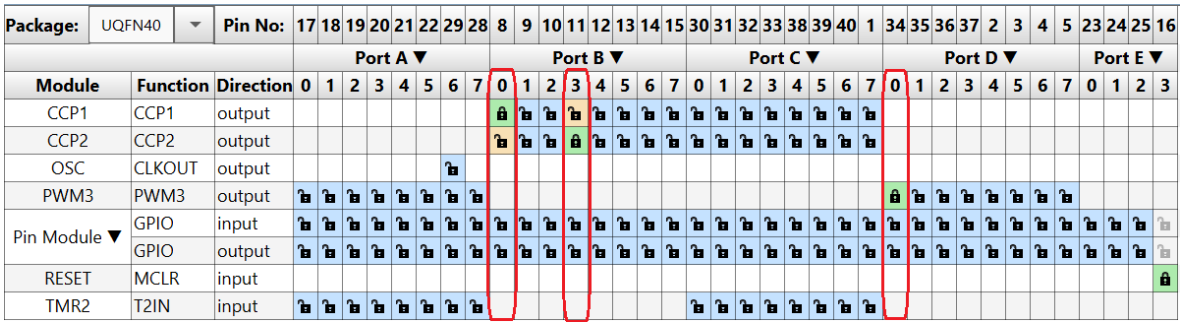
4. From the Device Resources window, add TMR2, CCP1, CCP2 and PWM3. Do the following configurations for each peripheral:
Timer2 Configuration:
   - **Hardware Settings** tab
     - Enable Timer: checked
     - Control Mode: Roll over pulse
     - Start/Reset Option: Software control
   - **Timer Clock** tab
     - Clock Source: $F_{OSC}/4$
     - Prescaler: 1:8
     - Postscaler: 1:16
   - **Timer Period** tab
     - Timer Period: 2.048 ms
   - **Software Settings** tab
     - Enable Timer Interrupt: checked

   CCP1 Configuration:
   - Enable CCP: checked
   - CCP Mode: PWM
   - Select Timer: Timer2
   - Duty Cycle: 100.0%
   - CCPR Alignment: left_aligned

   CCP2 Configuration:
   - Enable CCP: checked
   - CCP Mode: PWM
   - Select Timer: Timer2
   - Duty Cycle: 100.0%
   - CCPR Alignment: left_aligned

   PWM3 Configuration:
   - Enable CCP: checked
   - Select Timer: Timer2
   - Duty Cycle: 100.0%
   - PWM Polarity: active_hi
5. Open *Pin Manager → Grid View* window, select **UQFN40** in the Package field and do the following pin configurations:
   - Set Port B pin 0 (RB0) as output for CCP1
   - Set Port B pin 3 (RB3) as output for CCP2
   - Set Port D pin 0 (PD0) as output for PWM3

Figure 4-2. Pin Mapping



6. Go to *Project Resources → System → Pin Module → Easy Setup* and enable WPU for the RE2 pin.
7. Click **Generate** in the **Project Resources** tab.
8. In the `main.c` file generated by MCC, add the following code:
   - Enable the global interrupt function
   - Add TMR2 interrupt function
   - Set the TMR2 interrupt handler initializer

```
void RGB_LED_Handler(void);

void main(void)
{
    /* Initialize the device */
    SYSTEM_Initialize();
    TMR2_SetInterruptHandler(RGB_LED_Handler);

    /* Enable the Global Interrupts */
    INTERRUPT_GlobalInterruptEnable();

    /* Enable the Peripheral Interrupts */
    INTERRUPT_PeripheralInterruptEnable();

    while (1)
    {
        //Add your application code
    }
}
```

The `RGB_LED_Handler()` function creates the color game on the RGB LED, as presented in the description of this use case.

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

4.2 Bare Metal Code

The necessary code and functions to implement the presented example are analyzed in this section.

The first step is to configure the microcontroller to disable the Watchdog Timer and to enable Low-Voltage Programming.

```
#pragma config WDTE = OFF
#pragma config LVP = ON
```

The internal oscillator must be set to the desired value. This example uses the HFINTOSC with a frequency of 64 MHz. This translates into the following function:

```
static void CLK_Initialize(void)
{
    /* Configure NOSC HFINTOSC; NDIV 1; FOSC = 64MHz */
    OSCCON1bits.NOSC = 6;
    OSCCON1bits.NDIV = 0;

    /* HFFRQ 64_MHz */
    OSCFRQbits.HFFRQ = 8;
}
```

The output driver for I/O pins RB0, RB3 and RD0 must be enabled. They will be used as output for CCP1, CCP2 and PWM3, respectively. This translates into the following function:

```
static void PORT_Initialize(void)
{
    /* RB0 is output for PWM1 */
```

```
    TRISBbits.TRISB0 = 0;
    /* RB3 is output for PWM2 */
    TRISBbits.TRISB3 = 0;
    /* RD0 is output for PWM3 */
    TRISDbits.TRISD0 = 0;
}
```

Timer2 uses as clock source $F_{OSC}/4$ with a 1:8 clock prescaler and a 1:16 postscaler and has the overflow interrupt enabled. This translates into the following function:

```
#define MAX_DCY                 1023

static void TMR2_Initialize(void)
{
    /* TIMER2 clock source is FOSC/4 */
    T2CLKCONbits.CS = 1;

    /* TIMER2 counter reset */
    T2TMR = 0x00;

    /* TIMER2 prescaler 1:8, postscaler 1:16 */
    T2CONbits.CKPS = 3;
    T2CONbits.OUTPS = 15;

    /* TIMER2 period register setting, divided by 4 because FOSC/4 is used for PWM mode */
    T2PR = MAX_DCY >> 2;

    /* Clearing IF flag before enabling the interrupt */
    PIR4bits.TMR2IF = 0;

    /* Enabling TIMER2 interrupt */
    PIE4bits.TMR2IE = 1;

    /* TIMER2 ON */
    T2CONbits.ON = 1;
}
```

CCP1 is configured in PWM mode and uses Timer2 as period generator. The value from the CCPR1 register is left-aligned and the initial duty cycle value is set at 100%, corresponding to a minimum brightness in common anode LEDs. For a common cathode LED, the initial duty cycle will be set at 0%. This translates into the following function:

```
#define MAX_DCY                         1023
#define LED_MIN_BRIGHT                  (MAX_DCY)

static void PWM1_Initialize(void)
{
    /* MODE PWM; EN enabled; FMT left_aligned */
    CCP1CONbits.MODE = 0x0C;
    CCP1CONbits.FMT = 1;
    CCP1CONbits.EN = 1;

    /* Selecting Timer 2 */
    CCPTMRSbits.C1TSEL = 1;

    /* Configure initial duty cycle */
    CCPR1 = (uint16_t)LED_MIN_BRIGHT << 6;
}
```

CCP2 is configured in PWM mode and uses Timer2 as period generator. The value from the CCPR2 register is left-aligned and the initial duty cycle value is set at 100%, corresponding to a minimum brightness in common anode LEDs. For a common cathode LED, the initial duty cycle will be set at 0%. This translates into the following function:

```
#define MAX_DCY                         1023
#define LED_MIN_BRIGHT                  (MAX_DCY)

static void PWM2_Initialize(void)
{
    /* MODE PWM; EN enabled; FMT left_aligned */
    CCP2CONbits.MODE = 0x0C;
    CCP2CONbits.FMT = 1;
    CCP2CONbits.EN = 1;
```

```
    /* Selecting Timer 2 */
    CCPTMRSbits.C2TSEL = 1;

    /* Configure initial duty cycle */
    CCPR2 = (uint16_t)LED_MIN_BRIGHT << 6;
}
```

PWM3 uses Timer2 as period generator. The initial duty cycle value is set at 100%, corresponding to a minimum brightness in common anode LEDs. For a common cathode LED, the initial duty cycle will be set at 0%. This translates into the following function:

```
#define MAX_DCY                         1023
#define LED_MIN_BRIGHT                  (MAX_DCY)

static void PWM3_Initialize(void)
{
    /* PWM3 enabled module, polarity normal */
    PWM3CONbits.EN = 1;
    PWM3CONbits.POL = 0;

    /* Select timer 2 */
    CCPTMRSbits.P3TSEL = 1;

    /* Configure initial duty cycle */
    PWM3DC = (uint16_t)LED_MIN_BRIGHT << 6;
}
```

Configuring the location of the pins is independent of the application purpose and the CCP mode. Each microcontroller has its own default physical pin position for peripherals, but the pin positions can be changed using the Peripheral Pin Select (PPS).

Therefore, the CCP pins can be relocated using the CCP1PPS and CCP2PPS registers for the input channels while using the RxyPPS registers for the output channels. The PPS configuration values can be found in the *Peripheral Pin Select Module* section of a device data sheet.

For this example, the output PWM signal from CCP1 will be routed to the RB0 pin, the output PWM signal from CCP2 will be routed to RB3, and the output PWM signal from PWM3 will be routed to RD0. This translates into the following code:

```
static void PPS_Initialize(void)
{
    /* Configure RB0 for PWM1 output */
    RB0PPS = 0x05;
    /* Configure RB3 for PWM2 output */
    RB3PPS = 0x06;
    /* Configure RD0 for PWM3 output */
    RD0PPS = 0x07;
}
```

Before any processing is done, the interrupts of the microcontroller must be activated. This is done by setting the Global Interrupt Enable (GIE) and the Peripheral Interrupt Enable (PIE) bits of the INTCON register.

```
static void  INTERRUPT_Initialize(void)
{
    /* Enable the Global Interrupts */
    INTCONbits.GIE = 1;

    /* Enable the Peripheral Interrupts */
    INTCONbits.PEIE = 1;
}
```

Whenever Timer2 interrupt occurs, the duty cycle of the PWM signals is changed as mentioned in the use case description.

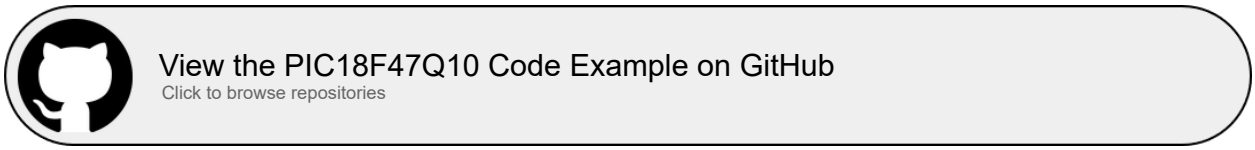
```
static void TMR2_ISR(void)
{
    /* clear the TMR2 interrupt flag */
    PIR4bits.TMR2IF = 0;

    /* user ISR function call; */
```

```
    RGB_LED_Handler();
}
```

The `RGB_LED_Handler()` function creates the color game on the RGB LED, as presented in the use case description. Using `LED_COMMON_ANODE` macro definition, the color sequence can be configured at compile time for either common anode or common cathode LEDs.

The Timer2 interrupt handler is handled by the following function:

```
static void __interrupt() INTERRUPT_InterruptManager (void)
{
    /* interrupt handler */
    if(INTCONbits.PEIE)
    {
        if( (PIE4bits.TMR2IE) && (PIR4bits.TMR2IF) )
        {
            TMR2_ISR();
        }
    }
}
```



5. References

1. MPLAB Code Configurator User's Guide
2. PIC1000: Getting Started with Writing C-Code for PIC16 and PIC18

6. Revision History

Doc Rev.	Date	Comments
B	10/2020	Updated PIC1000 link from References section
A	5/2020	Initial document release

The Microchip Website

Microchip provides online support via our website at www.microchip.com/. This website is used to make files and information easily available to customers. Some of the content available includes:

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQs), technical support requests, online discussion groups, Microchip design partner program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

Product Change Notification Service

Microchip's product change notification service helps keep customers current on Microchip products. Subscribers will receive email notification whenever there are changes, updates, revisions or errata related to a specified product family or development tool of interest.

To register, go to www.microchip.com/pcn and follow the registration instructions.

Customer Support

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Embedded Solutions Engineer (ESE)
- Technical Support

Customers should contact their distributor, representative or ESE for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in this document.

Technical support is available through the website at: www.microchip.com/support

Microchip Devices Code Protection Feature

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specifications contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is secure when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods being used in attempts to breach the code protection features of the Microchip devices. We believe that these methods require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Attempts to breach these code protection features, most likely, cannot be accomplished without violating Microchip's intellectual property rights.
- Microchip is willing to work with any customer who is concerned about the integrity of its code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of its code. Code protection does not mean that we are guaranteeing the product is "unbreakable." Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Legal Notice

Trademarks

ISBN: 978-1-5224-6880-6

Quality Management System

For information regarding Microchip's Quality Management Systems, please visit www.microchip.com/quality.

Worldwide Sales and Service

AMERICAS	ASIA/PACIFIC	ASIA/PACIFIC	EUROPE
Corporate Office 2355 West Chandler Blvd. Chandler, AZ 85224-6199 Tel: 480-792-7200 Fax: 480-792-7277 Technical Support: www.microchip.com/support Web Address: www.microchip.com **Atlanta** Duluth, GA Tel: 678-957-9614 Fax: 678-957-1455 **Austin, TX** Tel: 512-257-3370 **Boston** Westborough, MA Tel: 774-760-0087 Fax: 774-760-0088 **Chicago** Itasca, IL Tel: 630-285-0071 Fax: 630-285-0075 **Dallas** Addison, TX Tel: 972-818-7423 Fax: 972-818-2924 **Detroit** Novi, MI Tel: 248-848-4000 **Houston, TX** Tel: 281-894-5983 **Indianapolis** Noblesville, IN Tel: 317-773-8323 Fax: 317-773-5453 Tel: 317-536-2380 **Los Angeles** Mission Viejo, CA Tel: 949-462-9523 Fax: 949-462-9608 Tel: 951-273-7800 **Raleigh, NC** Tel: 919-844-7510 **New York, NY** Tel: 631-435-6000 **San Jose, CA** Tel: 408-735-9110 Tel: 408-436-4270 **Canada - Toronto** Tel: 905-695-1980 Fax: 905-695-2078	**Australia - Sydney** Tel: 61-2-9868-6733 **China - Beijing** Tel: 86-10-8569-7000 **China - Chengdu** Tel: 86-28-8665-5511 **China - Chongqing** Tel: 86-23-8980-9588 **China - Dongguan** Tel: 86-769-8702-9880 **China - Guangzhou** Tel: 86-20-8755-8029 **China - Hangzhou** Tel: 86-571-8792-8115 **China - Hong Kong SAR** Tel: 852-2943-5100 **China - Nanjing** Tel: 86-25-8473-2460 **China - Qingdao** Tel: 86-532-8502-7355 **China - Shanghai** Tel: 86-21-3326-8000 **China - Shenyang** Tel: 86-24-2334-2829 **China - Shenzhen** Tel: 86-755-8864-2200 **China - Suzhou** Tel: 86-186-6233-1526 **China - Wuhan** Tel: 86-27-5980-5300 **China - Xian** Tel: 86-29-8833-7252 **China - Xiamen** Tel: 86-592-2388138 **China - Zhuhai** Tel: 86-756-3210040	**India - Bangalore** Tel: 91-80-3090-4444 **India - New Delhi** Tel: 91-11-4160-8631 **India - Pune** Tel: 91-20-4121-0141 **Japan - Osaka** Tel: 81-6-6152-7160 **Japan - Tokyo** Tel: 81-3-6880- 3770 **Korea - Daegu** Tel: 82-53-744-4301 **Korea - Seoul** Tel: 82-2-554-7200 **Malaysia - Kuala Lumpur** Tel: 60-3-7651-7906 **Malaysia - Penang** Tel: 60-4-227-8870 **Philippines - Manila** Tel: 63-2-634-9065 **Singapore** Tel: 65-6334-8870 **Taiwan - Hsin Chu** Tel: 886-3-577-8366 **Taiwan - Kaohsiung** Tel: 886-7-213-7830 **Taiwan - Taipei** Tel: 886-2-2508-8600 **Thailand - Bangkok** Tel: 66-2-694-1351 **Vietnam - Ho Chi Minh** Tel: 84-28-5448-2100	**Austria - Wels** Tel: 43-7242-2244-39 Fax: 43-7242-2244-393 **Denmark - Copenhagen** Tel: 45-4485-5910 Fax: 45-4485-2829 **Finland - Espoo** Tel: 358-9-4520-820 **France - Paris** Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79 **Germany - Garching** Tel: 49-8931-9700 **Germany - Haan** Tel: 49-2129-3766400 **Germany - Heilbronn** Tel: 49-7131-72400 **Germany - Karlsruhe** Tel: 49-721-625370 **Germany - Munich** Tel: 49-89-627-144-0 Fax: 49-89-627-144-44 **Germany - Rosenheim** Tel: 49-8031-354-560 **Israel - Ra'anana** Tel: 972-9-744-7705 **Italy - Milan** Tel: 39-0331-742611 Fax: 39-0331-466781 **Italy - Padova** Tel: 39-049-7625286 **Netherlands - Drunen** Tel: 31-416-690399 Fax: 31-416-690340 **Norway - Trondheim** Tel: 47-72884388 **Poland - Warsaw** Tel: 48-22-3325737 **Romania - Bucharest** Tel: 40-21-407-87-50 **Spain - Madrid** Tel: 34-91-708-08-90 Fax: 34-91-708-08-91 **Sweden - Gothenberg** Tel: 46-31-704-60-40 **Sweden - Stockholm** Tel: 46-8-5090-4654 **UK - Wokingham** Tel: 44-118-921-5800 Fax: 44-118-921-5820