TB3285

Getting Started with Timers/Counters on PIC18

Introduction

Authors: Alin Stoicescu, Marius Nicolae, Stefan Vlad, Microchip Technology Inc.

This technical brief provides information about the Timers/Counters present on the PIC18 families of microcontrollers.

The document describes the application area, the modes of operation and the hardware and software requirements of the Timers/Counters and configurable output or input for internal or external use with the help of the Peripheral Pin Select (PPS).

Throughout the document, the configuration of the used peripherals for each use case will be described in detail. Additionally, this technical brief explains the concepts of the TMR0, TMR1/3/5 and TMR2/4/6 and their implementation in the PIC18 family of microcontrollers with the following use cases:

1. TIMER 0
   1.1. **Using TMR0 in 8-bit Mode with Periodic Interrupt**:
   This example describes how to configure TMR0 in 8-bit mode and generate a compare interrupt every 100 ms, using LFINTOSC as clock source. A GPIO pin is toggled each time an interrupt occurs.
   1.2. **Using and operating TMR0 in 16-bit Mode while the Microcontroller is in Sleep**:
   This example describes how to configure and operate TMR0 in 16-bit mode while the microcontroller is in Sleep mode and generate an overflow interrupt every ten seconds. When the interrupt occurs, a GPIO pin connected to an LED is ON for 100 ms and then the microcontroller is put back to Sleep.
   1.3. **Using TMR0 in 8-bit Mode and to Generate an Output Signal**:
   This example describes how to configure TMR0 in 8-bit mode and generate a 125 Hz signal on one of the T0 output pins using Peripheral Pin Select (PPS).
2. TIMER 1/3/5
   2.1. **Using TMR1 Gate to Measure Frequency:**
   This example shows how to use the TMR1 configured in Gate Single-Pulse and Toggle Combined mode. It will sample a full period of a signal. A GPIO pin will be configured as input and it will be connected to a periodical signal.
   2.2. **Using TMR1 to Trigger a Special Event**:
   This example shows how to use the TMR1 configured as a counter. The Capture/Compare/PWM (CCP) module will be configured with a user-defined value. A GPIO pin will be configured as an output for the CCP. When the counter reaches the CCP value, the pin logic value will be toggled.
   2.3. **Using TMR1 Gate to Measure Short vs Long Button Press**:
   This example shows how to use the TMR1 configured in Gate Single-Pulse mode. It will start counting when the button is pressed. Two different interrupts will be activated based on how long the button was pressed.
3. TIMER 2/4/6
   3.1. **Using TMR2 as Auto-conversion Trigger for ADCC Module**
   This example will present how to use TMR2 peripheral to trigger the ADCC to make conversions at a fixed frequency rate that can be adjusted by modifying the period of TMR2.
   3.2. **Using TMR4 in One-Shot Mode with External Signal as Reset**
   This example will present how to use TMR4 peripheral in One-Shot mode to stop TMR2 if an external pin is pulled to GND for more than the desired period.
   3.3. **Using TMR4 as HLT to Generate an Interrupt (like a WDT without Reset)**

This example will present how to use the TMR4 as a Hardware Limit Timer (HLT) in order to generate an interrupt and stop TMR2 that also stops the ADCC auto-conversion.

3.4. **Using TMR2 as Alternate SPI Clock**
This example will present how to use the TMR2 as alternate clock for SPI peripheral with a 10 kHz frequency.

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

View Code Examples on GitHub
Click to browse repositories

Table of Contents

1. Peripheral Overview

TIMER0

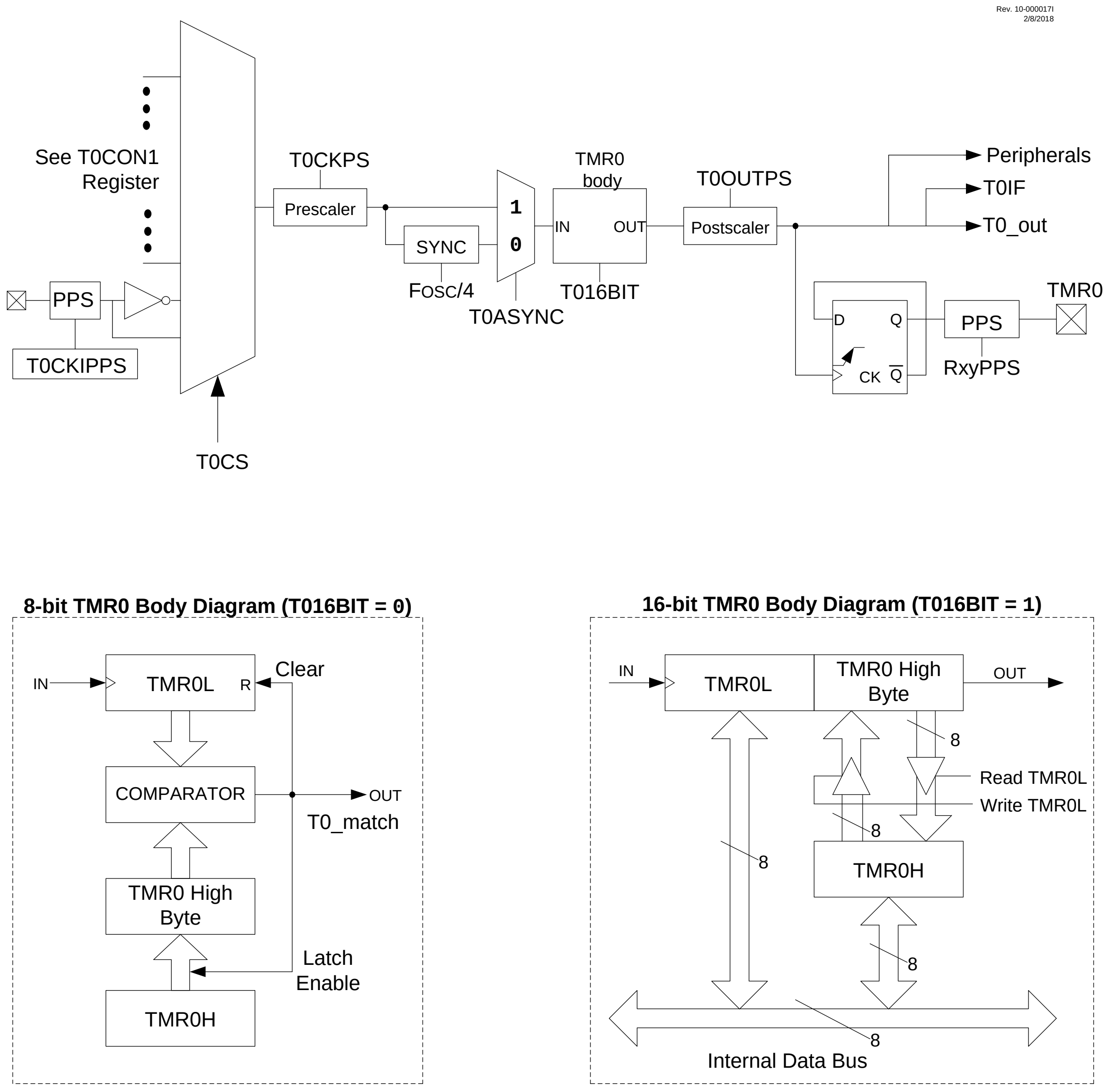
Timer0 can operate either as an 8-bit or 16-bit timer. The 16-bit mode is enabled by setting the T016BIT bit.

In the 8-bit mode, a buffered version of TMR0H is maintained. This is compared with the value of TMR0L on each cycle of the selected clock source. When the two values match, the following events occur:

- TMR0L is reset
- The contents of TMR0H are copied to the TMR0H buffer for next comparison

In the 16-bit mode, TMR0H:TMR0L form the 16-bit timer value and read and write of the TMR0H register are buffered. Timer0 rolls over to 0x0000 on incrementing past 0xFFFF. This makes the timer free-running. TMR0L/H registers cannot be reloaded in this mode once started. In both 8-bit and 16-bit modes, Timer0 increments on the rising edge of the selected clock source.

Figure 1-1. Timer0 Block Diagram



TIMER 1/3/5

Timer1 module is a 16-bit incrementing counter. When used with an internal clock source, the module is a timer and increments on every instruction cycle. When used with an external clock source, the module can be used either as a timer or counter and increments on every selected edge of the external source. Timer1 can function on several possible synchronous and asynchronous clock sources. When the FOSC internal clock source is selected, the Timer1 register value will increment by four counts every instruction clock cycle. Due to this condition, a 2-LSB error in resolution will occur when reading the Timer1 value. To utilize the full resolution of Timer1, an asynchronous input signal must be used to gate the Timer1 clock input.

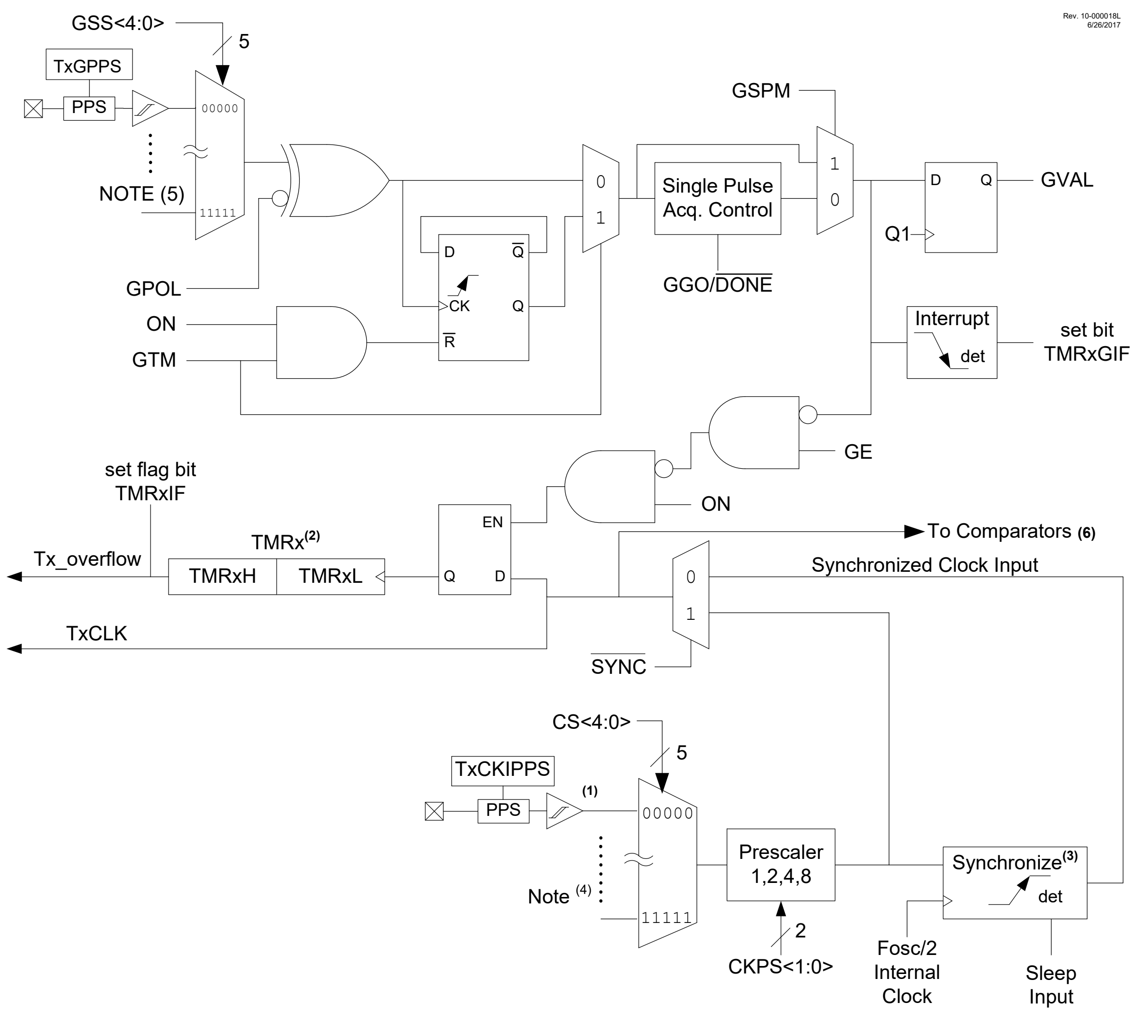
Important: References to module Timer1 apply to all the odd numbered timers on this device.

Timer1 is a 16-bit module which has the following features:

- 16-Bit Timer/Counter register
- Optionally synchronized comparator out
- Multiple Timer1 gate (count enable) sources
- Interrupt-on-Overflow
- Wake-Up on Overflow (external clock, Asynchronous mode only)
- Time base for the capture/compare function with the CCP modules
- Special Event Trigger (with CCP)

The following figure is a simplified diagram showing signal flow through the TMR1.

Figure 1-2. Timer1 Block Diagram



TIMER 2/4/6

Timer2 operates in three major modes:

- Free Running Period
- One-shot
- Monostable

Free-Running Period Mode

The value of T2TMR is compared to that of the Period register (T2PR) on each clock cycle. When the two values match, the comparator resets the value of T2TMR to 00h on the next cycle and increments the output postscaler counter. When the postscaler count equals the value in the OUTPS bits of the T2CON register, then a one clock period wide pulse occurs on the TMR2_postscaled output and the postscaler count is cleared.

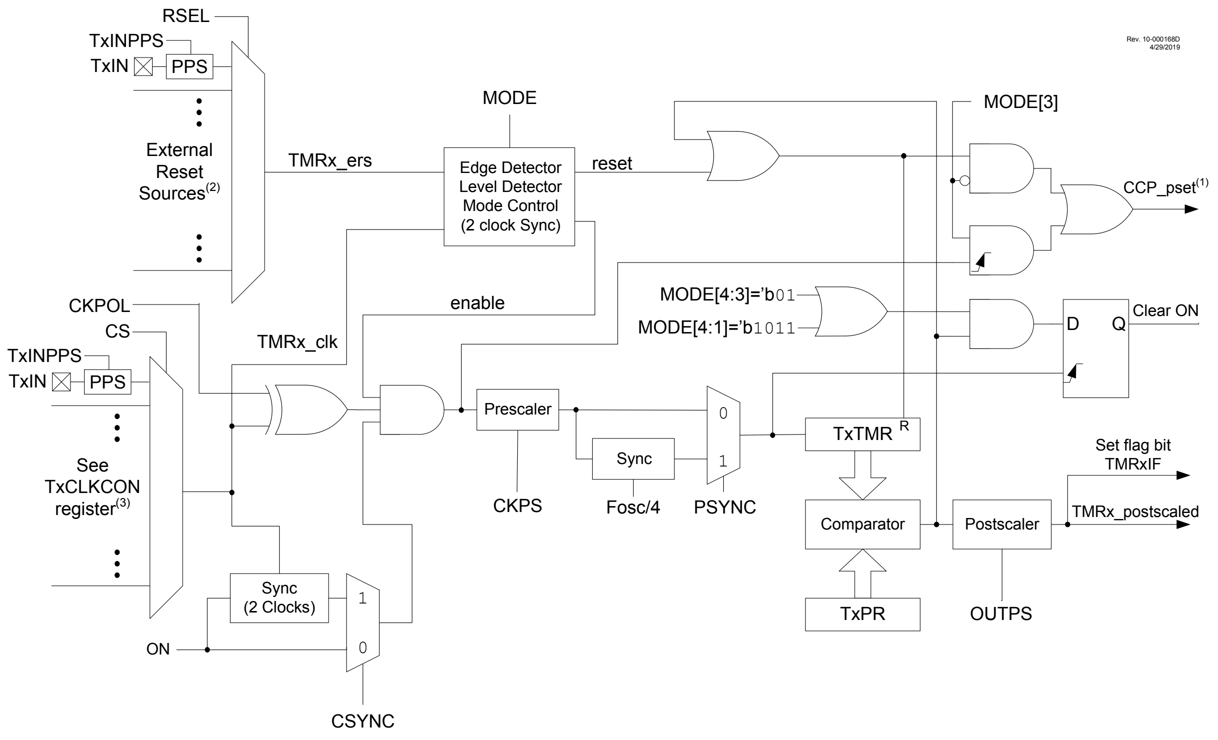
One-Shot Mode

The One-Shot mode is identical to the Free-Running Period mode except for when the ON bit is cleared and the timer is stopped when T2TMR matches T2PR and will not restart until the ON bit is cycled off and on. The Postscaler (OUTPS) values other than zero are ignored in this mode because the timer is stopped at the first period event and the postscaler is reset when the timer is restarted.

Monostable Mode

Monostable modes are similar to One-Shot modes except for when the ON bit is not cleared and the timer can be restarted by an external Reset event.

Figure 1-3. Timer2 Block Diagram



Notes:

1. This signal comes from the pin selected by TxCKIPPS.
2. TMRx register increments on rising edge.
3. Synchronize does not operate while in Sleep.
4. See TMRxCLK for clock source selections from device data sheet.
5. See TMRxGATE for gate source selection from device data sheet.
6. Synchronized comparator output should not be used in conjunction with synchronized input clock.

2. Using TMR0 in 8-bit Mode with Periodic Interrupt

This example describes how to configure Timer0 in 8-bit mode and to generate a compare interrupt every 100 ms using LFINTOSC as clock source. A GPIO pin (the development board's on-board LED) will be configured as output and toggled each time the interrupt occurs. Additionally, the main clock will use a separate clock source (HFINTOSC) and Timer0 will run asynchronously from the main clock.

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

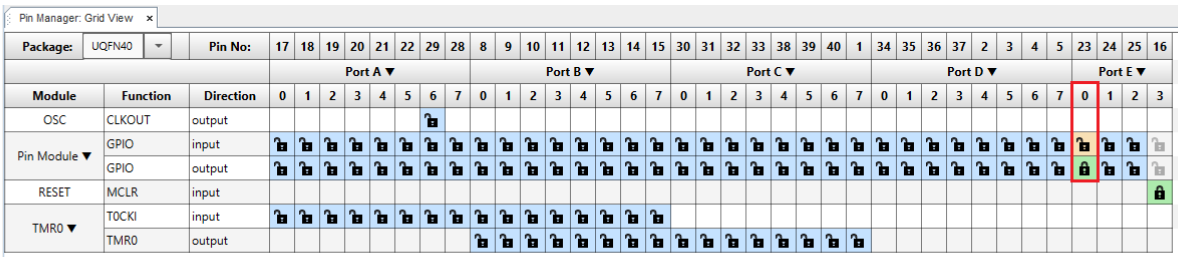
- System clock initialization
- Port initialization
- Timer0 initialization
- Interrupts initialization
- Timer0 interrupt handling

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 MCC from the toolbar (more information about how to install the MCC plug-in can be found here).
3. Go to *Project Resources → System → System Module* and make the following configurations:
   – Oscillator Select: HFINTOSC
   – HF Internal Clock: 1 MHz
   – Clock Divider: 1
   – In the Watchdog Timer Enable field in the **WWDT** tab, **WDT Disabled** has to be selected
   – In the **Programming** tab, **Low-Voltage Programming Enable** has to be checked
4. From the Device Resources window, add TMR0 and do the following configurations:
   Timer0 Configuration:
   – Enable Timer: checked
   – **Timer Clock** tab
     - Clock Source: LFINTOSC
     - Clock Prescaler: 1:16
     - Postscaler: 1:1
     - Timer mode: 8-bit
     - Enable Synchronization: unchecked
   – Timer period: 100 ms
   – Enable Timer Interrupt: checked
5. Open *Pin Manager → Grid View* window and select **UQFN40** in the MCU package field and make the following pin configuration:
   – Set Port E pin 0 (RE0) as output

Figure 2-1. Pin Mapping


Pin Manager: Grid View

Package:	UQFN40	Pin No:	17	18	19	20	21	22	29	28	8	9	10	11	12	13	14	15	30	31	32	33	38	39	40	1	34	35	36	37	2	3	4	5	23	24	25	16
			Port A ▼								Port B ▼								Port C ▼								Port D ▼								Port E ▼			
Module	Function	Direction	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3
OSC	CLKOUT	output							🔓																													
Pin Module ▼	GPIO	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	GPIO	output	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔒	🔓	🔓	🔓
RESET	MCLR	input																																				🔒
TMR0 ▼	T0CKI	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓																				
	TMR0	output									🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓												

6. Click **Pin Module** in the Project Resources and set the custom name for RE0 to LED0.
7. Click **Generate** in the **Project Resources** tab.

8. In the `main.c` file generated by MCC, change or add the following code:
   – Enable the global and peripheral interrupts
   – Add the TMR0 Interrupt function
   – Set the TMR0 interrupt handler initializer

```
void TMR0_compareInterrupt(void);

void main(void)
{
    // Initialize the device
    SYSTEM_Initialize();

    // Enable the Global Interrupts
    INTERRUPT_GlobalInterruptEnable();

    // Enable the Peripheral Interrupts
    INTERRUPT_PeripheralInterruptEnable();

    TMR0_SetInterruptHandler(TMR0_compareInterrupt);

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

void TMR0_compareInterrupt(void)
{
    LED0_Toggle();
}
```

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

2.2 Bare Metal Code

First, the Watchdog Timer has to be disabled and Low-Voltage Programming (LVP) has to be enabled using the following pragma code:

```
#pragma config WDTE = OFF   /* WDT operating mode->WDT Disabled */
#pragma config LVP = ON     /* Low voltage programming enabled, RE3 pin is MCLR */
```

The following function initializes the system clock to have the HFINTOSC oscillator as input clock and to run at 1 MHz:

```
static void CLK_Initialize(void)
{
    OSCCON1 = 0x60;    /* set HFINTOSC as new oscillator source */
    OSCFRQ = 0x00;     /* set HFFRQ to 1 MHz */
}
```

The following function initializes the RE0 pin (corresponding to the on-board LED0) as output pin:

```
static void PORT_Initialize(void)
{
    TRISEbits.TRISE0 = 0;   /* configure RE0 as output */
}
```

The following function initializes Timer0 in 8-bit mode, sets the prescaler to 1:16, loads TMR0H and TMR0L registers, clears the Interrupt flag, and enables the interrupt and Timer0:

```
static void TMR0_Initialize(void)
{
    T0CON1 = 0x94;          /* Select LFINTOSC, set the prescaler to 1:16, Disable TMR0 sync */
    TMR0H = 0xC1;           /* Load the compare value to TMR0H */
    TMR0L = 0x00;           /* Load the reset value to TMR0L */
    PIR0bits.TMR0IF = 0;    /* clear the interrupt flag */
    PIE0bits.TMR0IE = 1;    /* enable TMR0 interrupt */
    T0CON0 = 0x80;          /* Configure TMR0 in 8-bit mode and enable TMR0 */
}
```

The following function enables the global and peripheral interrupts:

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

The following function handles the Timer0 interrupt and it is called in the interrupt manager function:

```
static void TMR0_ISR(void)
{
    PIR0bits.TMR0IF = 0;                 /* clear the TMR0 interrupt flag */
    LATEbits.LATE0 = ~LATEbits.LATE0;    /* toggle LED0 */
}
```

The following function handles the interrupts in the project:

```
void __interrupt() INTERRUPT_InterruptManager (void)
{
    /* Check if TMR0 interrupt is enabled and if the interrupt flag is set */
    if(PIE0bits.TMR0IE == 1 && PIR0bits.TMR0IF == 1)
    {
        TMR0_ISR();
    }
}
```

The main function will call all the initializing functions and run all the peripherals in an infinite empty loop:

```
void main(void)
{
    CLK_Initialize();
    PORT_Initialize();
    TMR0_Initialize();
    INTERRUPT_Initialize();

    while(1)
    {
        ;
    }
}
```

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

3. Using and Operating TMR0 in 16-bit Mode while the Microcontroller is in Sleep

This example describes how to configure TMR0 in 16-bit mode and generate an overflow interrupt every ten seconds, using LFINTOSC as clock source. TMR0 will run while the microcontroller is in Sleep mode. A GPIO pin (the development board's on-board LED) will be configured as output. When the interrupt occurs, the microcontroller is woken up and the LED is lit for 100 ms and then the microcontroller is put back to Sleep.

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

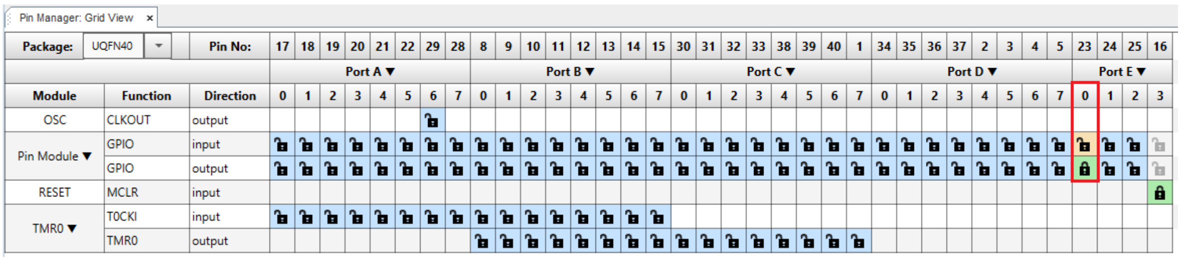
- System clock initialization
- Port initialization
- Timer0 initialization
- Interrupts initialization
- Timer0 interrupt handling

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 MCC from the toolbar (more information about how to install the MCC plug-in can be found here).
3. Go to *Project Resources → System → System Module* and make the following configurations:
   - Oscillator Select: HFINTOSC
   - HF Internal Clock: 1 MHz
   - Clock Divider: 1
   - In the Watchdog Timer Enable field in the **WWDT** tab, **WDT Disabled** has to be selected
   - In the **Programming** tab, **Low-Voltage Programming Enable** has to be checked
4. From the Device Resources window, add TMR0 and do the following configurations:
   Timer0 Configuration:
   - Enable Timer: checked
   - **Timer Clock** tab
     - Clock Source: LFINTOSC
     - Clock Prescaler: 1:32
     - Postscaler: 1:1
     - Timer mode: 16-bit
     - Enable Synchronization: unchecked
   - Timer period: 10s
   - Enable Timer Interrupt: checked
5. Open *Pin Manager → Grid View* window and select **UQFN40** in the MCU package field and make the following pin configuration:
   - Set Port E pin 0 (RE0) as output

Figure 3-1. Pin Mapping


Pin Manager: Grid View ×

Package:	UQFN40	Pin No:	17	18	19	20	21	22	29	28	8	9	10	11	12	13	14	15	30	31	32	33	38	39	40	1	34	35	36	37	2	3	4	5	23	24	25	16
			Port A ▼								Port B ▼								Port C ▼								Port D ▼								Port E ▼			
Module	Function	Direction	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3
OSC	CLKOUT	output							🔓																													
Pin Module ▼	GPIO	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	GPIO	output	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔒	🔓	🔓	🔓
RESET	MCLR	input																																				🔒
TMR0 ▼	T0CKI	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓																				
	TMR0	output									🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓												

6. Click **Pin Module** in the Project Resources and set the custom name for RE0 to LED0.
7. Click **Generate** in the **Project Resources** tab.
8. In the `main.c` file generated by MCC, change or add the following code:
   – Enable the Global and Peripheral interrupts
   – Light up LED0, wait 100 ms, turn off LED0 and put the microcontroller to Sleep
9.
```
void main(void)
{
    // Initialize the device
    SYSTEM_Initialize();

    // Enable the Global Interrupts
    INTERRUPT_GlobalInterruptEnable();

    // Enable the Peripheral Interrupts
    INTERRUPT_PeripheralInterruptEnable();

    while (1)
    {
        LED0_SetLow();
        __delay_ms(100);
        LED0_SetHigh();
        SLEEP();
    }
}
```

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

3.2 Bare Metal Code

First, the Watchdog Timer has to be disabled and Low-Voltage Programming (LVP) has to be enabled using the following pragma code:

```
#pragma config WDTE = OFF   /* WDT operating mode->WDT Disabled */
#pragma config LVP = ON     /* Low voltage programming enabled, RE3 pin is MCLR */
```

The following function initializes the system clock to have the HFINTOSC oscillator as input clock and to run at 1 MHz:

```
static void CLK_Initialize(void)
{
    OSCCON1 = 0x60;    /* set HFINTOSC as new oscillator source */
    OSCFRQ = 0x00;     /* set HFFRQ to 1 MHz */
}
```

The following function initializes the RE0 pin (corresponding to the on-board LED0) as output pin:

```
static void PORT_Initialize(void)
{
    TRISEbits.TRISE0 = 0;   /* configure RE0 as output */
}
```

The following function initializes Timer0 in 16-bit mode, sets the prescaler to 1:32, loads the TMR0H and TMR0L registers, clears the Interrupt flag and enables the interrupt and Timer0:

```
static void TMR0_Initialize(void)
{
    T0CON1 = 0x95;         /* select LFINTOSC, disable TMR0 sync, set prescaler to 1:32 */
    TMR0H = 0xDA;          /* set TMR0H reload value */
    TMR0L = 0x29;          /* set TMR0L reload value */
    PIR0bits.TMR0IF = 0;   /* clear the interrupt flag */
    PIE0bits.TMR0IE = 1;   /* enable TMR0 interrupt */
    T0CON0 = 0x90;        /* configure TMR0 in 16-bit mode and enable TMR0 */
}
```

The following function enables the Global and Peripheral interrupts:

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

The following function handles the Timer0 interrupt and it is called in the Interrupt Manager function:

```
static void TMR0_ISR(void)
{
    PIR0bits.TMR0IF = 0;    /* clear the TMR0 interrupt flag */
    TMR0H = 0xDA;           /* set TMR0H reload value */
    TMR0L = 0x29;           /* set TMR0L reload value */
}
```

The following function handles the interrupts in the project:

```
void __interrupt() INTERRUPT_InterruptManager(void)
{
    /* Check if TMR0 interrupt is enabled and if the interrupt flag is true */
    if(PIE0bits.TMR0IE == 1 && PIR0bits.TMR0IF == 1)
    {
        TMR0_ISR();
    }
}
```

The main function will call all the initializing functions and will turn on the LED0 for 100 ms and put the microcontroller to Sleep using the `SLEEP()` instruction. Additionally, prior to the main function, the _XTAL_FREQ symbol must be defined and set to 1000000 (equivalent to the 1 MHz system frequency) for the use of the `__delay_ms()` function:

```
#define _XTAL_FREQ                  1000000UL

void main(void)
{
    CLK_Initialize();
    PORT_Initialize();
    TMR0_Initialize();
    INTERRUPT_Initialize();

    while(1)
    {
        LATEbits.LATE0 = 0;    /* turn LED ON */
        __delay_ms(100);       /* wait 100 ms */
        LATEbits.LATE0 = 1;    /* turn LED OFF */

        SLEEP();
    }
}
```

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

4. Using TMR0 in 8-bit Mode and to Generate an Output Signal

This example describes how to configure TMR0 in 8-bit mode, using LFINTOSC as clock source. A GPIO pin will be configured as output and a 125 Hz signal will be generated on the GPIO pin using the Peripheral Pin Select (PPS).

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

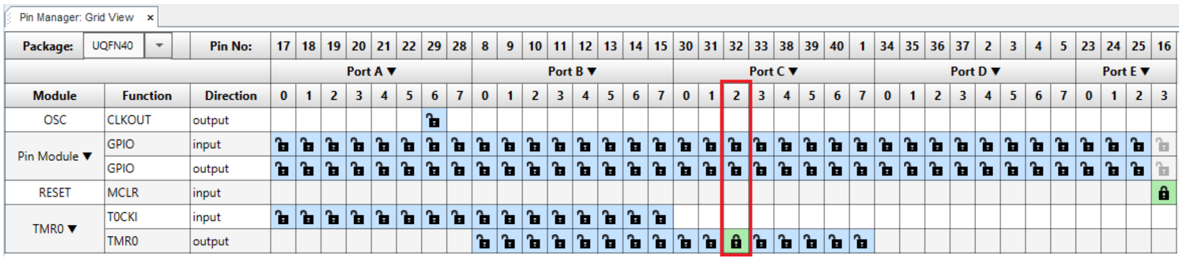
- System clock initialization
- Port initialization
- Timer0 initialization
- Peripheral Pin Select initialization

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 MCC from the toolbar (more information about how to install the MCC plug-in can be found here).
3. Go to *Project Resources → System → System Module* and make the following configurations:
   – Oscillator Select: HFINTOSC
   – HF Internal Clock: 1 MHz
   – Clock Divider: 1
   – In the Watchdog Timer Enable field in the **WWDT** tab, **WDT Disabled** has to be selected
   – In the **Programming** tab, **Low-Voltage Programming Enable** has to be checked
4. From the Device Resources window, add TMR0 and make the following configurations:
   Timer0 Configuration:
   – Enable Timer: checked
   – **Timer Clock** tab
     • Clock Source: LFINTOSC
     • Clock Prescaler: 1:1
     • Postscaler: 1:1
     • Timer mode: 8-bit
     • Enable Synchronization: unchecked
   – Timer period: 4 ms
   – Enable Timer Interrupt: unchecked
5. Open *Pin Manager → Grid View* window and select **UQFN40** in the MCU package field and make the following pin configuration:
   – Set Port C pin 2 (RE0) as output

Figure 4-1. Pin Mapping



Package:	UQFN40	Pin No:	17	18	19	20	21	22	29	28	8	9	10	11	12	13	14	15	30	31	32	33	38	39	40	1	34	35	36	37	2	3	4	5	23	24	25	16
			Port A ▼								Port B ▼								Port C ▼								Port D ▼								Port E ▼			
Module	**Function**	**Direction**	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3
OSC	CLKOUT	output							🔓																													
Pin Module ▼	GPIO	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	GPIO	output	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
RESET	MCLR	input																																				🔒
TMR0 ▼	T0CKI	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓																				
	TMR0	output									🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔒	🔓	🔓	🔓	🔓	🔓												

6. Click **Generate** in the **Project Resources** tab.

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

4.2 Bare Metal Code

First, the Watchdog Timer has to be disabled and Low-Voltage Programming (LVP) has to be enabled using the following pragma code:

```
#pragma config WDTE = OFF   /* WDT operating mode->WDT Disabled */
#pragma config LVP = ON     /* Low voltage programming enabled, RE3 pin is MCLR */
```

The following function initializes the system clock to have the HFINTOSC oscillator as input clock and to run at 1 MHz:

```
static void CLK_Initialize(void)
{
    OSCCON1 = 0x60;    /* set HFINTOSC as new oscillator source */
    OSCFRQ = 0x00;     /* set HFFRQ to 1 MHz */
}
```

The following function initializes the RC2 pin as output pin:

```
static void PORT_Initialize(void)
{
    TRISCbits.TRISC2 = 0;   /* configure RC2 as output */
}
```

The following function initializes Timer0 in 8-bit mode, sets the prescaler to 1:1, loads the TMR0H and TMR0L registers, clears the Interrupt flag and enables Timer0:

```
static void TMR0_Initialize(void)
{
    T0CON1 = 0x90;          /* select LFINTOSC and disable TMR0 sync*/
    TMR0H = 0x7B;           /* load TMR0H */
    TMR0L = 0x00;           /* load TMR0L */
    PIR0bits.TMR0IF = 0;    /* clear the interrupt flag */
    T0CON0 = 0x80;          /* enable TMR0 */
}
```

The following function configures the TMR0 output to RC2 in PPS:

```
static void PPS_Initialize(void)
{
    RC2PPS = 0x13;      /* configure RC2 for TMR0 output */
}
```

The main function calls all the initializing functions and run all the peripherals in an infinite empty loop:

```
void main(void)
{
    CLK_Initialize();
    PORT_Initialize();
    TMR0_Initialize();
    PPS_Initialize();

    while(1)
    {
        ;
    }
}
```

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

5. Using TMR1 Gate to Measure Frequency

This example describes how to initialize and use the TMR1 in Gate-Single Pulse and Toggle combined mode. The timer will start counting on an incrementing edge, will measure a full-cycle length of a gate signal and will stop when a new incrementing edge appears. An interrupt will be generated when the measurement is completed. A GPIO pin will be configured as input and the periodical signal will be applied on this pin.

In this example, the microcontroller was configured with a clock system of 32 MHz and the timer was configured with a clock source frequency of 1 MHz and is able to measure the following range of values:

- The smallest frequency value: This is based on the number of values that the timer can count. It is a 16-bit timer so it can count up to 65,535, resulting in a frequency of approximately 15.26 Hz.
- The biggest frequency value: This is based on the Nyquist frequency theorem. The sampling frequency must be at least two times bigger than the one of the measured signal to obtain a more accurate result. This results in a frequency of approximately 500 kHz.

Note: It is recommended to increase the clock source frequency of the timer to measure frequencies closer or bigger than the Nyquist value from the above example.

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

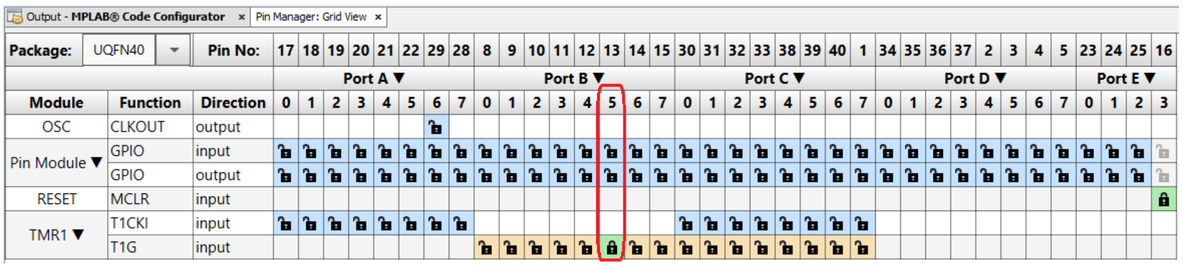
- System clock initialization
- Port initialization
- Timer1 initialization
- Interrupts initialization
- Timer1 gate interrupt handling

5.1 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 MCC from the toolbar (more information about how to install the MCC plug-in can be found here).
3. Go to *Project Resources → System → System Module* and make the following configurations:
   – Oscillator Select: HFINTOSC
   – HF Internal Clock: 32 MHz
   – Clock Divider: 1
   – In the Watchdog Timer Enable field in the **WWDT** tab, **WDT Disabled** has to be selected.
   – In the **Programming** tab, **Low-Voltage Programming Enable** has to be checked
4. From the Device Resources window, add TMR1 and make the following configurations:
   Timer1 Configuration:
   – Enable Timer: checked
   – **Timer Clock** tab
      - Clock Source: FOSC/4
      - Prescaler: 1:8
   – **Enable Gate** tab: checked
      - Enable Gate Toggle: checked
      - Enable Gate Single-Pulse mode: checked
   – Enable Timer Gate Interrupt: checked
5. Open *Pin Manager → Grid View* window and select **UQFN40** in the MCU package field and make the following pin configurations to enable the internal signal access to the I/O:

Figure 5-1. Pin Mapping



Package:	UQFN40	Pin No:	17	18	19	20	21	22	29	28	8	9	10	11	12	13	14	15	30	31	32	33	38	39	40	1	34	35	36	37	2	3	4	5	23	24	25	16
			Port A ▼								Port B ▼								Port C ▼								Port D ▼								Port E ▼			
Module	Function	Direction	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3
OSC	CLKOUT	output							🔓																													
Pin Module ▼	GPIO	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	GPIO	output	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
RESET	MCLR	input																																				🔒
TMR1 ▼	T1CKI	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓									🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓												
	T1G	input									🔓	🔓	🔓	🔓	🔓	🔒	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓												

6. Click **Generate** in the **Project Resources** tab.
7. For this example, some extra code is required aside from the one generated from MCC.
   – The Global and Peripheral interrupts need to be enabled in the `main.c` file. The macros were created by the MCC and the user needs to remove the "//" so they are no longer treated as comments:

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

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

   – In the `tmr1.c` file, the `TMR1_GATE_ISR()` function needs to be updated to clear the Interrupt flag, read the counted value, reset it afterward and re-enable the timer gate control for a new acquisition. The following configuration is used:

```
void TMR1_GATE_ISR(void)
{
    volatile uint16_t value = 0;

    PIR5 &= ~(_PIR5_TMR1GIF_MASK);

    value = TMR1_ReadTimer();

    TMR1_WriteTimer(0);

    T1GCON |= _T1GCON_T1GGO_MASK;
}
```

Note: To obtain the frequency of the measured signal from the counted value read, the clock source frequency of the timer needs to be divided by the value.

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

5.2 Bare Metal Code

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

The first step will be to configure the microcontroller to disable the Watchdog Timer and to enable Low-Voltage Programming (LVP).

```
#pragma config WDTE = OFF /* WDT operating mode->WDT Disabled */
#pragma config LVP = ON /* Low voltage programming enabled, RE3 pin is MCLR */
```

As described in the example functionality, an initialization of peripherals must be added to the project: TMR1, the system clock, the GPIO pin and the interrupts.

The system clock was configured to use the HFINTOSC oscillator with an internal frequency of 32 MHz. The following function is used:

```
/* Clock initialization function */
static void CLK_Initialize(void)
{
    /* set HFINTOSC as new oscillator source */
    OSCCON1bits.NOSC = 0x6;

    /* set HFFRQ to 32MHz */
    OSCFRQbits.HFFRQ = 0x6;
}
```

The GPIO peripheral was configured to use PINB5 as input for the signal that needs to be measured. The following function is used:

```
/* Port initialization function */
static void PORT_Initialize(void)
{
    /* configure RB5 as input */
    TRISBbits.TRISB5 = 1;

    /* configure RB5 as digital */
    ANSELBbits.ANSELB5 = 0;
}
```

The TMR1 peripheral was configured in Gate Single-Pulse and Toggle combined mode, has a clock source of 1 MHz, the counter is active on a trailing edge and the peripheral's gate interrupt is Active. The following function is used:

```
/* TMR1 initialization function */
static void TMR1_Initialize(void)
{
    /* Timer controlled by gate function */
    T1GCONbits.GE = 1;

    /* Timer gate toggle mode enabled */
    T1GCONbits.GTM = 1;

    /* Timer gate active high */
    T1GCONbits.GPOL = 1;

    /* Timer acquistion is ready */
    T1GCONbits.GGO_nDONE = 1;

    /* Timer gate single pulse mode enabled */
    T1GCONbits.T1GSPM = 1;

    /* Source Clock FOSC/4 */
    T1CLKbits.CS = 0x1;

    /* Clearing gate IF flag before enabling the interrupt */
    PIR5bits.TMR1GIF = 0;

    /* Enabling TMR1 gate interrupt */
    PIE5bits.TMR1GIE = 1;

    /* CLK Prescaler 1:8 */
    T1CONbits.CKPS = 0x3;

    /* TMR1 enabled */
    T1CONbits.ON = 1;
}
```

The microcontroller's interrupts were enabled and are used to determine when the signal measurement is done. The following function is used:

```
/* Interrupt initialization function */
static void INTERRUPT_Initialize(void)
{
    /* Enable the Global Interrupts */
    INTCONbits.GIE = 1;
    /* Enable the Peripheral Interrupts */
    INTCONbits.PEIE = 1;
}
```

When the timer finishes measuring the frequency of the external signal, an interrupt will occur in the interrupt manager. It will check for the source of the interrupt and, if it is from TMR1 gate, will call a handler function. The following function is used:

```
/* Interrupt handler function */
static void __interrupt() INTERRUPT_InterruptManager(void)
{
    // interrupt handler
    if(INTCONbits.PEIE == 1)
    {
        if(PIE5bits.TMR1GIE == 1 && PIR5bits.TMR1GIF == 1)
        {
            TMR1_GATE_ISR();
        }
        else
        {
            //Unhandled Interrupt
        }
    }
    else
    {
        //Unhandled Interrupt
    }
}
```

The handler needs to clear the Interrupt flag, read the counted value and reset it afterward, and re-enable the timer gate control for a new acquisition. The following function is used:

```
/* TMR1 gate ISR function */
static void TMR1_GATE_ISR(void)
{
    volatile uint16_t value = 0;

    /* Clearing gate IF flag */
    PIR5bits.TMR1GIF = 0;

    /* Read TMR1 value */
    value = TMR1_readTimer();

    /* Reset the counted value */
    TMR1_writeTimer(0);

    /* Prepare for next read */
    T1GCONbits.GGO_nDONE = 1;
}

static uint16_t TMR1_readTimer(void)
{
    /* Return TMR1 value */
    return ((uint16_t)TMR1H << 8) | TMR1L;
}

static void TMR1_writeTimer(uint16_t timerValue)
{
    /* Write TMR1H value */
    TMR1H = timerValue >> 8;

    /* Write TMR1L value */
    TMR1L = timerValue;
}
```

Note: To obtain the frequency of the measured signal from the counted value read, the clock source frequency of the timer needs to be divided by the value.

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

6. Using TMR1 to Trigger a Special Event

This example describes how to initialize and use the TMR1 as a counter. The Capture/Compare/PWM (CCP) module will be configured with a user-defined value. A GPIO pin will be configured as output and the event will toggle the logic value of this pin. The event will be triggered when the counted value from TMR1 will be equal with the CCP value.

In this example, the microcontroller was configured with a clock system of 1 MHz and the timer was configured with a clock source frequency of 250 kHz. It is a 16-bit timer so it can count up to 65,535. The CCP value was set to 4,095 in this example. When the counter reaches this value, an event will occur which will be strictly handled by the hardware peripheral, without any software and load on the core.

The event can be configured to clear or not clear the timer counter value and, if the GPIO pin should be set high, set low or toggled every time the event is triggered. In this example, the event will toggle the GPIO pin and will not clear the timer, so the timer counted value will overflow when reaches the 65,535 maxim value and will restart counting from zero. Thus, even when a value was predefined for CCP, the event will be triggered with a frequency of 250 kHz / 65,535 ~ = 3.81 Hz.

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

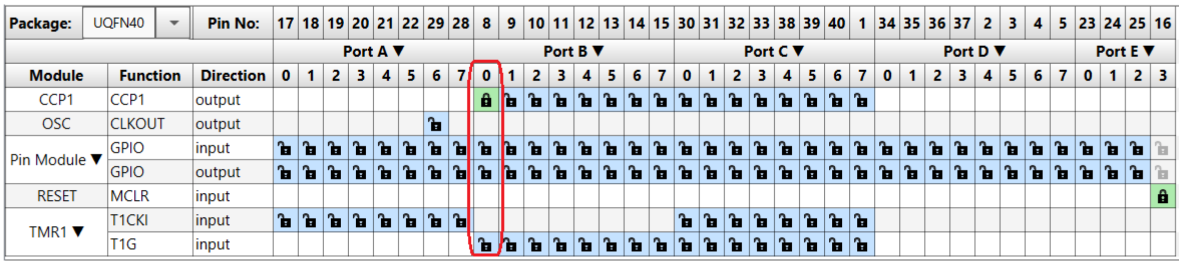
- System clock initialization
- PPS initialization
- Port initialization
- Timer1 initialization
- CCP initialization

6.1 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 MCC from the toolbar (more information about how to install the MCC plug-in can be found here).
3. Go to *Project Resources → System → System Module* and make the following configurations:
   – Oscillator Select: HFINTOSC
   – HF Internal Clock: 32 MHz
   – Clock Divider: 32
   – In the Watchdog Timer Enable field in the **WWDT** tab, **WDT Disabled** has to be selected.
4. From the Device Resources window, add TMR1 and CCP1. Make the following configurations for each peripheral:
   – TMR1 Configuration:
      1. Enable Timer: checked
      2. **Timer Clock** tab
         – Clock Source: FOSC/4
         – Prescaler: 1:1
         – Enable Synchronization: Checked
   – CCP1 Configuration:
      - Enable CCP: Checked
      - **CCP Mode** tab: Compare
         – Select Timer: Timer1
         – Compare Mode: Toggle_
5. Open *Pin Manager → Grid View* window and select **UQFN40** in the MCU package field and make the following pin configuration to enable the internal signal access to the I/O:

Figure 6-1. Pin Mapping



Package:	UQFN40	Pin No:	17	18	19	20	21	22	29	28	8	9	10	11	12	13	14	15	30	31	32	33	38	39	40	1	34	35	36	37	2	3	4	5	23	24	25	16
			Port A ▼								Port B ▼								Port C ▼								Port D ▼								Port E ▼			
Module	**Function**	**Direction**	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3
CCP1	CCP1	output									🔒	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓												
OSC	CLKOUT	output							🔓																													
Pin Module ▼	GPIO	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	GPIO	output	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
RESET	MCLR	input																																				🔒
TMR1 ▼	T1CKI	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓									🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓												
	T1G	input									🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓												

6. Click **Generate** in the **Project Resources** tab.

Note: In this example, the event will be strictly handled by the hardware peripheral without any software, and load on the core so no extra code was used aside from the one generated from MCC.

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

6.2 Bare Metal Code

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

The first step will be to configure the microcontroller to disable the Watchdog Timer and to enable Low-Voltage Programming (LVP).

```
#pragma config WDTE = OFF /* WDT operating mode->WDT Disabled */
#pragma config LVP = ON /* Low voltage programming enabled, RE3 pin is MCLR */
```

As described in the example functionality, an initialization of peripherals must be added to the project: TMR1, the system clock and the GPIO pin.

The system clock was configured to use the HFINTOSC oscillator with an internal frequency of 32 MHz and the clock divided by 32, so the actual system frequency is 1 MHz. The following function is used:

```
/* Clock initialization function */
static void CLK_Initialize(void)
{
    /* set HFINTOSC as new oscillator source */
    OSCCON1bits.NOSC = 0x6;

    /* set Clock Div by 32 */
    OSCCON1bits.NDIV = 0x5;

    /* set HFRQ to 32MHz */
    OSCFRQbits.HFFRQ = 0x6;
}
```

The GPIO peripheral was configured to use PINB0 as output for the event triggered by CCP. The following function is used:

```
/* PPS initialization function */
static void PPS_Initialize(void)
{
    /* Configure RB0 for CCP1 output */
    RB0PPS = 0x05;
}

/* Port initialization function */
static void PORT_Initialize(void)
{
    /* Set RB0 as output */
    TRISBbits.TRISB0 = 0;
}
```

The TMR1 peripheral was configured as a normal counter. The following function is used:

```
/* TMR1 initialization function */
static void TMR1_Initialize(void)
{
    /* Set timer Source Clock to FOSC/4 */
    T1CLKbits.CS = 0x1;

    /* Enable timer */
    T1CONbits.ON = 1;
}
```

The CCP1 peripheral was configured to toggle a GPIO pin when the TMR1 counted value is equal with the CCP value. The following function is used:

```
/* CCP1 initialization function */
static void CCP1_Initialize(void)
{
    /* Select TMR1 as input for CCP1*/
    CCPTMRSbits.C1TSEL = 0x1;

    /* Set the high value for compare */
    CCPR1H = 0x0F;

    /* Set the low value for compare */
    CCPR1L = 0xFF;

    /* Compare mode with toggle*/
    CCP1CONbits.CCP1MODE = 0x2;

    /* Enable CCP1 */
    CCP1CONbits.EN = 1;
}
```

Note: In this example, the event will be strictly handled by the hardware peripheral without any software and load on the core.

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

7. Using TMR1 Gate to Measure Short vs. Long Button Press

This example describes how to initialize and use the TMR1 in Gate Single-Pulse mode. The timer will start counting on an falling edge. If the leading edge appears, a gate interrupt will be generated, denoting that the button was short pressed. If the timer overflows before the leading edge appears, an overflow interrupt will be generated, denoting that the button was long pressed. A GPIO pin will be configured as input and connected to a button.

Note: The polarity of the gate is based on the button logic. If the button is active-low (meaning it will provide zero logic value when pressed), the timer needs to count on negative polarity and start counting on falling edge.

In this example, the microcontroller was configured with a clock system of 1 MHz and the timer was configured with a clock source frequency of 31,250 MHz = 32 μs and is able to measure the following range of values:

- The smallest pressed time: This is based on the clock frequency of the timer, resulting in a minimum time of 1 / 31,250 Hz = 32 μs.
- The biggest pressed time: This is based on the maximum value the timer can count. It is a 16-bit timer so it can count up to 65,535. Thus, resulting in a minimum time of 32 μs * 65,535 ~ = 2.1 s. A longer press will result in a timer overflow.

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

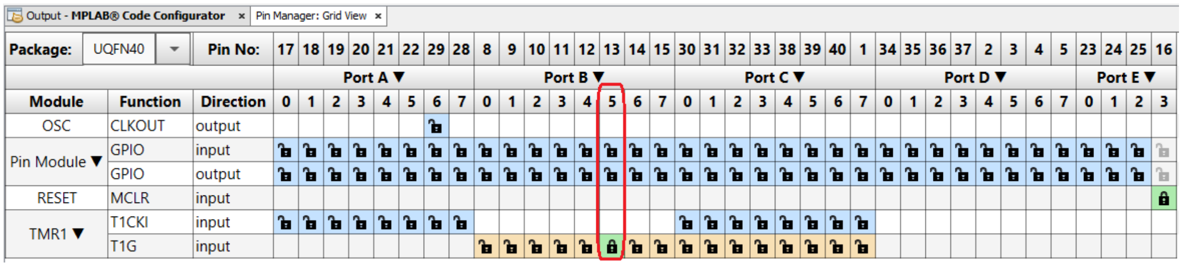
- System clock initialization
- Port initialization
- Timer1 initialization
- Interrupts initialization
- Timer1 interrupt handling
- Timer1 gate interrupt handling

7.1 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 MCC from the toolbar (more information about how to install the MCC plug-in can be found here).
3. Go to *Project Resources → System → System Module* and make the following configurations:
   – Oscillator Select: HFINTOSC
   – HF Internal Clock: 32 MHz
   – Clock Divider: 32
   – In the Watchdog Timer Enable field, in the **WWDT** tab, **WDT Disabled** has to be selected.
4. From the Device Resources window, add TMR1 and do the following configurations:
   – Enable Timer: checked
   – **Timer Clock** tab
      • Clock Source: FOSC/4
      • Prescaler: 1:8
   – **Enable Gate** tab: checked
      • Enable Gate Toggle: checked
      • Enable Gate Single-Pulse mode: checked
      • Gate Polarity: Low
   – Enable Timer Interrupt: Checked
   – Enable Timer Gate Interrupt: Checked
5. Open *Pin Manager → Grid View* window and select **UQFN40** in the MCU package field and make the following pin configuration to enable the internal signal access to the I/O:

Figure 7-1. Pin Mapping


Output - MPLAB® Code Configurator | Pin Manager: Grid View

Package:	UQFN40	Pin No:	17	18	19	20	21	22	29	28	8	9	10	11	12	13	14	15	30	31	32	33	38	39	40	1	34	35	36	37	2	3	4	5	23	24	25	16
			Port A ▼								Port B ▼								Port C ▼								Port D ▼								Port E ▼			
Module	**Function**	**Direction**	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3
OSC	CLKOUT	output							🔓																													
Pin Module ▼	GPIO	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	GPIO	output	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
RESET	MCLR	input																																				🔒
TMR1 ▼	T1CKI	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓									🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓												
	T1G	input									🔓	🔓	🔓	🔓	🔓	🔒	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓												

6. Click **Generate** in the **Project Resources** tab.
7. For this example, some extra code is required aside from the one generated from MCC.
   – The Global and Peripheral interrupts need to be enabled in the `main.c` file. The macros were created by the MCC and the user needs to remove the "//" so they are no longer treated as comments:

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

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

   – In the `tmr1.c` file, the `TMR1_ISR()` function needs to be updated to stop the gate control because the button was not released yet and it will generate an undesired interrupt when that will happen. It also needs to clear the Interrupt flag, reset the counted value and re-enable the timer gate control for a new acquisition. The following function is used:

```
void TMR1_ISR(void)

    T1GCON &= ~_T1GCON_T1GGO_MASK;

    PIR4 &= ~_PIR4_TMR1IF_MASK;

    PIR5 &= ~_PIR5_TMR1GIF_MASK;

    TMR1_WriteTimer(0);

    T1GCON |= _T1GCON_T1GGO_MASK;

    if(TMR1_InterruptHandler)
    {
        TMR1_InterruptHandler();
    }
}
```

   – In the `tmr1.c` file, the `TMR1_GATE_ISR()` function needs to be updated to clear the Interrupt flag, reset the counted value and re-enable the timer gate control for a new acquisition. The following function is used:

```
void TMR1_GATE_ISR(void)
{
    PIR5 &= ~(_PIR5_TMR1GIF_MASK);

    TMR1_WriteTimer(0);

    T1GCON |= _T1GCON_T1GGO_MASK;
}
```

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

7.2 Bare Metal Code

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

The first step will be to configure the microcontroller to disable the Watchdog Timer and to enable Low-Voltage Programming (LVP):

```
#pragma config WDTE = OFF /* WDT operating mode->WDT Disabled */
#pragma config LVP = ON /* Low voltage programming enabled, RE3 pin is MCLR */
```

As described in the example functionality, the following peripherals must be initialized: TMR1, the system clock, the GPIO pin and the interrupts.

The system clock was configured to use the HFINTOSC oscillator with an internal frequency of 32 MHz and the clock divided by 32, so the actual system frequency is 1 MHz. The following function is used:

```
/* Clock initialization function */
static void CLK_Initialize(void)
{
    /* set HFINTOSC as new oscillator source */
    OSCCON1bits.NOSC = 0x6;

    /* set Clock Div by 32 */
    OSCCON1bits.NDIV = 0x5;

    /* set HFFRQ to 32MHz */
    OSCFRQbits.HFFRQ = 0x6;
}
```

The GPIO peripheral was configured to use PINB5 as input for TMR1 button. The following function is used:

```
/* Port initialization function */
static void PORT_Initialize(void)
{
    /* configure RB5 as input */
    TRISBbits.TRISB5 = 1;

    /* configure RB5 as digital */
    ANSELBbits.ANSELB5 = 0;
}
```

The TMR1 peripheral is configured in Gate Single-Pulse mode, has a clock source of 1 MHz, the counter is active on a falling edge and the peripheral gate and overflow interrupts are Active. The following function is used:

```
/* TMR1 initialization function */
static void TMR1_Initialize(void)
{
    /* Timer controlled by gate function */
    T1GCONbits.GE = 1;

    /* Timer acquistion is ready */
    T1GCONbits.GGO_nDONE = 1;

    /* Timer gate single pulse mode enabled */
    T1GCONbits.T1GSPM = 1;

    /* Source Clock FOSC/4 */
    T1CLKbits.CS = 0x1;

    /* Clearing IF flag before enabling the interrupt */
    PIR4bits.TMR1IF = 0;

    /* Enabling TMR1 interrupt */
    PIE4bits.TMR1IE = 1;

    /* Clearing gate IF flag before enabling the interrupt */
    PIR5bits.TMR1GIF = 0;

    /* Enabling TMR1 gate interrupt */
    PIE5bits.TMR1GIE = 1;
    /* CLK Prescaler 1:8 */
    T1CONbits.CKPS = 0x3;

    /* TMR1 enabled */
    T1CONbits.ON = 1;
}
```

The microcontroller's interrupts were enabled and are used to determine the button press time. The following function is used:

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

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

When the timer finishes counting the button pressed time, an interrupt will occur in the interrupt manager. It will check for the source of the interrupt and it will call a handler function. The following function is used:

```
/* Interrupt handler function */
static void __interrupt() INTERRUPT_interruptManager(void)
{
    // interrupt handler
    if(INTCONbits.PEIE == 1)
    {
        if(PIE4bits.TMR1IE == 1 && PIR4bits.TMR1IF == 1)
        {
            TMR1_ISR();
        }
        else if(PIE5bits.TMR1GIE == 1 && PIR5bits.TMR1GIF == 1)
        {
            TMR1_GATE_ISR();
        }
        else
        {
            //Unhandled Interrupt
        }
    }
    else
    {
        //Unhandled Interrupt
    }
}
```

The overflow interrupt will occur when the button is pressed for so long that the timer maximum value is exceeded. The handler needs to stop the gate control because the button was not released yet and it will generate an undesired interrupt when that will happen. It also needs to clear the Interrupt flag, reset the counted value and re-enable the timer gate control for a new acquisition. The following function is used:

```
/* TMR1 ISR function */
static void TMR1_ISR(void)
{
    /* Stop Gate control */
    T1GCONbits.GGO_nDONE = 0;

    /* Clearing overflow IF flag */
    PIR4bits.TMR1IF = 0;

    /* Clearing gate IF flag */
    PIR5bits.TMR1GIF = 0;

    /* Reset the counted value */
    TMR1_writeTimer(0);

    /* Prepare for next read */
    T1GCONbits.GGO_nDONE = 1;
}
```

The gate interrupt will occur when the button is released before the timer reaches its maximum value. The handler needs to clear the Interrupt flag, reset the counted value and re-enable the timer gate control for a new acquisition. The following function is used:

```
/* TMR1 GATE ISR function */
static void TMR1_GATE_ISR(void)
{
    /* Clearing gate IF flag after button release */
    PIR5bits.TMR1GIF = 0;

    /* Reset the counted value */
    TMR1_writeTimer(0);

    /* Prepare for next read */
    T1GCONbits.GGO_nDONE = 1;
}
```

When any interrupt occurs, the timer counted value is reset to ‘0’. The following function is used:

```
static void TMR1_writeTimer(uint16_t timerValue)
{
    /* Write TMR1H value */
    TMR1H = timerValue >> 8;

    /* Write TMR1L value */
    TMR1L = timerValue;
}
```

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

8. Using TMR2 for Auto-Conversion Trigger for the ADCC Module

This example will present how to use the TMR2 peripheral to trigger the ADCC to make conversions at a fixed frequency rate that can be adjusted by modifying the TMR2 period.

The application will blink the LED0 with a rate of Timer2 period (100 ms), get the ADCC value and compare it with a desired threshold and, if it is higher, the LED0 will stop blinking.

This example uses the PIC18F47Q10 Curiosity Nano board with a POT click, both inserted into a Curiosity Nano adapter. For more details, visit the Hardware Configuration section in the GitHub repository.

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

- System clock initialization
- ADCC initialization and reading
- Port initialization
- Timer2 initialization
- Interrupts handling and initialization

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

8.1 MCC Generated Code

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

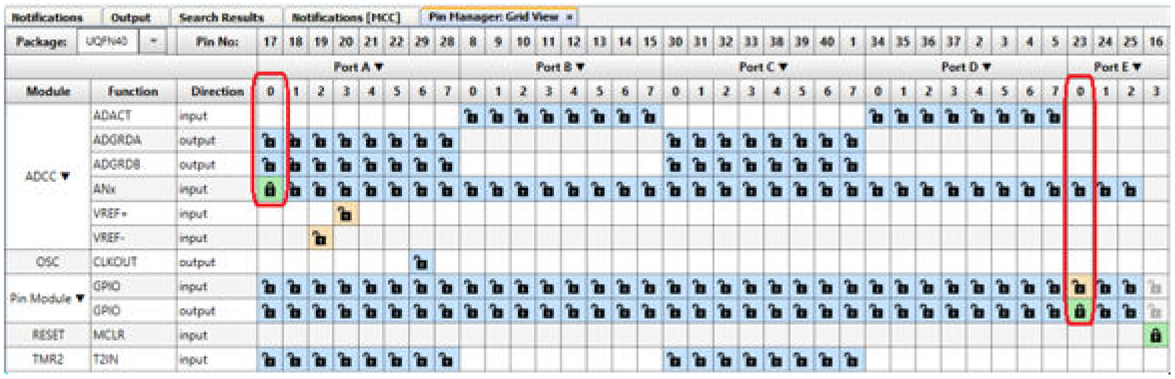
1. Create a new MPLAB X IDE project for PIC18F47Q10.
2. Open MCC from the toolbar (more information about how to install the MCC plug-in can be found here).
3. Go to *Project Resources → System → System Module* and make the following configurations:
   – Oscillator Select: HFINTOSC
   – HF Internal Clock: 1 MHz
   – Clock Divider: 1
   – In the Watchdog Timer Enable field in the **WWDT** tab, **WDT Disabled** has to be selected
   – In the **Programming** tab, **Low-Voltage Programming Enable** has to be checked
4. From the Device Resources window, add TMR2 and ADCC and then make the following configurations:
   Timer2 Configuration:
   – Enable Timer: checked
   – Control Mode: Roll over pulse
   – Start/Reset Option: Software control
   – **Timer Clock** tab
     - Clock Source: LFINTOSC
     - Clock Prescaler: 1:64
     - Postscaler: 1:1
   – Set 100 ms period in the **Timer Period** tab

   ADCC Configuration:
   – Enable ADCC: checked
   – Operating: Basic mode
   – In the **ADC** tab choose the following options:
     - *ADC Clock → Clock Source*: Select FRC
     - Auto-conversion Trigger: Select TMR2
   – **CVD Features** tab:

- Enable ADC Interrupt: checked

5. Open *Pin Manager → Grid View* window and select UQFN40 in the MCU package field and select the I/O pins outputs to enable the internal signal access to the I/O.

Figure 8-1. Pin Mapping



Notifications | Output | Search Results | Notifications [MCC] | Pin Manager: Grid View

Package: UQFN40		Pin No:	17	18	19	20	21	22	29	28	8	9	10	11	12	13	14	15	30	31	32	33	38	39	40	1	34	35	36	37	2	3	4	5	23	24	25	16
			Port A ▼								Port B ▼								Port C ▼								Port D ▼								Port E ▼			
Module	Function	Direction	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3
ADCC ▼	ADACT	input									🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓									🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓				
	ADGRDA	output	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓									🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓												
	ADGRDB	output	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓									🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓												
	ANx	input	🔒	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	
	VREF+	input				🔓																																
	VREF-	input			🔓																																	
OSC	CLKOUT	output							🔓																													
Pin Module ▼	GPIO	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	GPIO	output	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔒	🔓	🔓	🔓
RESET	MCLR	input																																				🔒
TMR2	T2IN	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓									🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓												

6. Click **Pin Module** in the Project Resources and set the custom name for RE0 to LED0, select **Output** box and for the ANA0 pin select the **Analog** box.
7. Click **Generate** in the **Project Resources** tab.
8. Add into the `main.c` file, the following lines of code:

```
#define DesiredThreshold    300      /* Desired threshold value */
volatile uint16_t adcVal;
static void ADCC_interrupt(void);

static void ADCC_interrupt(void)
{
    /* Toggle LED0 at the Timer2Period frequency */
    LED0_Toggle();
    adcVal = ADCC_GetConversionResult();
}

void main(void)
{
    // Initialize the device
    SYSTEM_Initialize();
ADCC_SetADIInterruptHandler(ADCC_Interrupt_by_TMR2);
    // Enable the Global Interrupts
    INTERRUPT_GlobalInterruptEnable();
    // Enable the Peripheral Interrupts
    INTERRUPT_PeripheralInterruptEnable();

    while (1)
    {
        if (adcVal > DesiredThreshold)
        {
            LED0_SetLow();
        }
    }
}
```

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

8.2 Bare Metal Code

The first step will be to configure the microcontroller to disable the Watchdog Timer and to enable Low-Voltage Programming (LVP):

```
#pragma config WDTE = OFF   /*disable Watchdog*/
#pragma config LVP = ON  /* Low voltage programming enabled, RE3 pin is MCLR */
```

Then, the following variables need to be defined:

```
#define Timer2Period        0x2F        /* TMR2 Period is 100ms */
#define DesiredThreshold    300         /* Desired threshold value */
#define AnalogChannel       0x00        /* Use ANA0 as input for ADCC */
volatile uint16_t adcVal;               /* ADCC global result value */
```

The `CLK_Initialize` function initializes the HFINTOSC internal oscillator:

```
static void CLK_Initialize(void)
{
    /* set HFINTOSC Oscillator */
    OSCCON1 = 0x60;
    /* set HFFRQ to 1 MHz */
    OSCFRQ = 0x00;
}
```

The `PORT_Initialize` function has the role to configure the pin used in this application, which is the RE0 output for LED0:

```
static void PORT_Initialize(void)
{
    /* Set RE0 digital input buffer disabled */
    ANSELE = 0x06;
    /* Set RE0 pin as output */
    TRISE = 0x06;
}
```

The next function initializes the ADCC and configures the TMR2 to be an auto-conversion trigger and enables the ADCC Interrupt flag:

```
static void ADCC_Initialize(void)
{
    /* ADACT Auto-Conversion Trigger Source is TMR2 */
    ADACT = 0x04;
    /* ADGO stop; ADFM right; ADON enabled; ADCONT disabled; ADCS FRC */
    ADCON0 = 0x94;
    /* Clear the ADCC interrupt flag */
    PIR1bits.ADIF = 0;
    /* Enabling ADCC interrupt flag */
    PIE1bits.ADIE = 1;
}
```

The Timer2 initialization function sets the clock source and the registers needed to generate an 100 ms period:

```
static void TMR2_Initialize(void)
{
    /* TMR2 Clock source, LFINTOSC (00100) has 31 kHz */
    T2CLKCON = 0x04;
    /* T2PSYNC Not Synchronized, T2MODE Software control, T2CKPOL Rising Edge */
    T2HLT = 0x00;
    /* TMR2ON on; T2CKPS Prescaler 1:64; T2OUTPS Postscaler 1:1
       Minimum timer period is 31 kHz/64 = 2.064516 ms  */
    T2CON = 0xE0;
    /* Set TMR2 period, PR2 to 100ms */
    T2PR = Timer2Period;
    /* Clear the TMR2 interrupt flag */
    PIR4bits.TMR2IF = 0;
}
```

The following initialization function will safely enable the Global and Peripherals interrupts, after they were initialized with proper settings:

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

The ADCC interrupt is triggered by TMR2 to complete a conversion at a frequency determined by the Timer2 Period. The following function handles the interrupts, in this case `ADCC_Interrupt`, checks the status of the ADCC Interrupt flag, and then calls the `ADCC_Interrupt` function:

```
static void __interrupt() INTERRUPT_InterruptManager(void)
{
    if (INTCONbits.PEIE == 1)
    {
        if (PIE1bits.ADIE == 1 && PIR1bits.ADIF == 1)
        {
            ADCC_Interrupt();
        }
    }
}
```

The `ADCC_Interrupt` function is separated from the interrupt manager to be similar to the code generated by MCC. In this function, the Interrupt flag is cleared first, the LED0 is toggled (this will happen with Timer2 Period frequency), and finally the ADCC value for the Analog Channel is read (ANA0 is used in this example).

```
static void ADCC_Interrupt(void)
{
    /* Clear the ADCC interrupt flag */
    PIR1bits.ADIF = 0;
    /* Toggle LED0 at the Timer2Period frequency */
    LATEbits.LATE0 = ~LATEbits.LATE0;
    /* Get the conversion result from ADCC AnalogChannel */
    adcVal = ADCC_ReadValue(AnalogChannel);
}
```

The ADCC read function only needs a parameter, the channel needed to be read:

```
static uint16_t ADCC_ReadValue(uint8_t channel)
{
    ADPCH = channel; /*Set the input channel for ADCC*/
    /* TMR2 is trigger source for auto-conversion for ADCC */
    return ((uint16_t)((ADRESH << 8) + ADRESL));
}
```

The next code in the `void main` function is an infinite loop (using a `while(1)`), which is used to check for the ADCC value using an “`if`” statement:

```
void main(void)
{
    /* Initialize the device */
    CLK_Initialize();             /* Oscillator Initialize function */
    PORT_Initialize();            /* Port Initialize function */
    ADCC_Initialize();            /* ADCC Initialize function */
    TMR2_Initialize();            /* TMR2 Initialize function */
    INTERRUPT_Initialize();       /* Interrupt Initialize function */

    while (1)
    {
        if (adcVal > DesiredThreshold)
        {
            /* turn LED0 ON by writing pin RE0 to low */
            LATEbits.LATE0 = 0;
        }
    }
}
```

This will check if the read value from Potentiometer (POT click) is above a defined value. If so, the LED0 will turn ON without blinking.

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

9. Using TMR4 in One-Shot Mode with External Signal as Reset

This example will present how to use the TMR4 peripheral in One-Shot mode to stop TMR2 if an external pin is pulled to GND for more than the desired period.

The application will blink the LED0 with a rate of Timer2 Period (100 ms) and, if the external pin RC7 is pulled down for more than the Timer4 Period (500 ms), the LED0 will stop blinking.

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

- System clock initialization
- Port initialization
- PPS initialization
- Timer2 initialization
- Timer4 initialization
- Interrupts handling and initialization

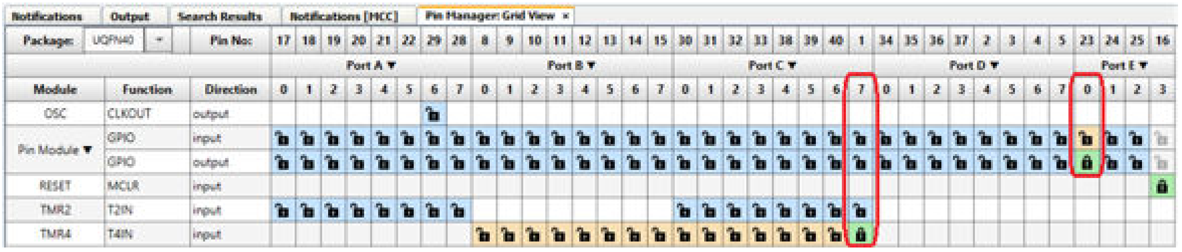
9.1 MCC Generated Code

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 MCC from the toolbar (more information about how to install the MCC plug-in can be found here).
3. Go to *Project Resources → System → System Module* and make the following configurations:
   - Oscillator Select: HFINTOSC
   - HF Internal Clock: 1 MHz
   - Clock Divider: 1
   - In the Watchdog Timer Enable field in the **WWDT** tab, **WDT Disabled** has to be selected
   - In the **Programming** tab, **Low-Voltage Programming Enable** has to be checked
4. From the Device Resources window, add TMR2, TMR4 and make the following configurations:
   Timer2 Configuration:
   - Enable Timer: checked
   - Control Mode: Roll over pulse
   - Ext. Reset Source: TMR4_postscaled
   - Start/Reset Option: Starts at T2ON = `1` and TMR2_ers = `0`
   - **Timer Clock** tab
     - Clock Source: LFINTOSC
     - Clock Prescaler: 1:64
     - Postscaler: 1:1
   - Set 100 ms period in the **Timer Period** tab
     - Enabled Timer Interrupt: checked
5. **Timer4 Configuration:**
   - Enable Timer: checked
   - Control mode: One shot
   - Ext. Reset Source: T4INPPS pin
   - Start/Reset Option: Starts at TMR4_ers = `0` and Resets at TMR4_ers = `1`
   - **Timer Clock** tab:
     - Clock Source: LFINTOSC
     - Clock Prescaler: 1:64
     - Postscaler: 1:1
   - Set 500 ms period in the **Timer Period** tab

6. Open *Pin Manager → Grid View* window, select UQFN40 in the MCU package field and select the I/O pins outputs to enable the internal signal access to the I/O.

Figure 9-1. Pin Mapping



Notifications | Output | Search Results | Notifications [MCC] | Pin Manager: Grid View ×

Package: UQFN40

Pin No:			17	18	19	20	21	22	29	28	8	9	10	11	12	13	14	15
			Port A ▼								Port B ▼							
Module	Function	Direction	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
OSC	CLKOUT	output							🔓									
Pin Module ▼	GPIO	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	GPIO	output	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
RESET	MCLR	input																
TMR2	T2IN	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓								
TMR4	T4IN	input									🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓

Pin No:			30	31	32	33	38	39	40	1	34	35	36	37	2	3	4	5	23	24	25	16
			Port C ▼								Port D ▼								Port E ▼			
Module	Function	Direction	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3
OSC	CLKOUT	output																				
Pin Module ▼	GPIO	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	GPIO	output	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔒	🔓	🔓	🔓
RESET	MCLR	input																				🔒
TMR2	T2IN	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓												
TMR4	T4IN	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔒												

7. Click **Pin Module** in the Project Resources, set the custom name for RE0 to LED0 and select **Output** box. For RC7 pin, select **WPU**.
8. Click **Generate** in the **Project Resources** tab.
9. The Interrupt function that will toggle LED0 at Timer2 period will need to be added before `main` function:

```
void TMR2_interrupt(void)
{
    /* Toggle LED0 at the Timer2Period frequency */
    LED0_Toggle();
}

void main(void)
{
    // Initialize the device
    SYSTEM_Initialize();
    TMR2_SetInterruptHandler(TMR2_interrupt);
    // Enable the Global Interrupts
    INTERRUPT_GlobalInterruptEnable();
    // Enable the Peripheral Interrupts
   INTERRUPT_PeripheralInterruptEnable();
    while (1)
    {
        // Add your application code
    }
}
```

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

9.2 Bare Metal Code

The first step will be to configure the microcontroller to disable the Watchdog Timer (WDT) and to enable Low-Voltage Programming (LVP).

```
#pragma config WDTE = OFF   /*disable Watchdog*/
#pragma config LVP = ON  /* Low voltage programming enabled, RE3 pin is MCLR */
```

The following constants need to be defined:

```
#define Timer2Period    0x2F        /* TMR2 Period is 100ms */
#define Timer4Period    0xF1        /* TMR4 Period is 500ms */
```

The `CLK_Initialize` function initializes the HFINTOSC internal oscillator:

```
static void CLK_Initialize(void)
{
    /* set HFINTOSC Oscillator */
    OSCCON1 = 0x60;
    /* set HFFRQ to 1 MHz */
    OSCFRQ = 0x00;
}
```

The `PPS_Initialize` function has the role to configure the RC7 peripheral select as input for TMR4:

```
static void PPS_Initialize(void)
{
    /* Set RC7 as input for TMR4 (T4IN) */
    T4INPPS = 0x17;
}
```

`PORT_Initialize` has the role to configure the RC7, input channel, and RE0 output for LED0 pins:

```
static PORT_Initialize(void)
{
    /* Set RC7 pin as digital */
    ANSELC = 0x7F;
    /* Set RE0 pin as output */
    TRISE = 0x06;
    /* Enable weak pull-up on pin RC7 */
    WPUC = 0x80;
}
```

The `TMR2_Initialize` function sets the clock source and the registers needed to generate an 100 ms period:

```
static void TMR2_Initialize(void)
{
    /* TMR2 Clock source, LFINTOSC (00100) has 31 kHz */
    T2CLKCON = 0x04;
    /* T2PSYNC Not Synchronized; T2MODE Starts at T2ON = 1 and TMR2_ers = 0; T2CKPOL Rising 
Edge */
    T2HLT = 0x02;
    /* TMR2 external reset is TMR4_postscaled */
    T2RST = 0x02;
    /* TMR2 ON on; T2 CKPS Prescaler 1:64; T2 OUTPS Postscaler 1:1
       Minimum timer period is 31 kHz/64 = 2.064516 ms  */
    T2CON = 0xE0;
    /* Set TMR2 period, PR2 to 100ms */
    T2PR = Timer2Period;
    /* Clear the TMR2 interrupt flag */
    PIR4bits.TMR2IF = 0;
    /* Enabling TMR2 interrupt */
    PIE4bits.TMR2IE = 1;
}
```

The `TMR4_Initialize` function sets the clock source and the registers needed to generate an 500 ms period:

```
static void TMR4_Initialize(void)
{
    /* TMR4 Clock source, LFINTOSC (00100) has 31 kHz */
    T4CLKCON = 0x04;
    /* TMR4 in OneShot mode, Starts at TMR4_ers=0 and resets on TMR4_ers=1 */
    T4HLT = 0x17;
    /* TMR4 External reset signal selected by T4INPPS pin  */
    T4RST = 0;
    /* TMR4 ON on; T4 CKPS Prescaler 1:64; T4 OUTPS Postscaler 1:1
       Minimum timer period is 31 kHz/64 = 2.064516 ms  */
    T4CON = 0xE0;
    /* Set TMR4 period, PR4 to 500ms */
    T4PR = Timer4Period;
    /* Clear the TMR4 interrupt flag */
    PIR4bits.TMR4IF = 0;
}
```

The following initialization function will safely enable the Global and Peripherals interrupts, after all modules have been initialized with proper settings:

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

The next function handles the interrupts (in this case there is only one interrupt), checks the status of the TMR2 Interrupt flag, and then calls the `TMR2_Interrupt` function.

```
static void __interrupt() INTERRUPT_InterruptManager(void)
{
    if (INTCONbits.PEIE == 1)
    {
        if (PIE4bits.TMR2IE == 1 && PIR4bits.TMR2IF == 1)
        {
            TMR2_Interrupt();
        }
    }
}
```

The `TMR2_Interrupt` clears the Interrupt flag and toggles the LED0 (this will happen with Timer2 Period frequency).

```
static void TMR2_Interrupt(void)
{
    /* Clear the TMR2 interrupt flag */
    PIR4bits.TMR2IF = 0;
    /* Toggle LED0 at the Timer2Period frequency */
    LATEbits.LATE0 = ~LATEbits.LATE0;
}
```

The `void main` function contains only the initialization functions:

```
void main(void)
{
    /* Initialize the device */
    CLK_Initialize();             /* Oscillator Initialize function */
    PPS_Initialize();             /* Peripheral select Initialize function */
    PORT_Initialize();            /* Port Initialize function */
    TMR2_Initialize();            /* TMR2 Initialize function */
    TMR4_Initialize();            /* TMR4 Initialize function */
    INTERRUPT_Initialize();       /* Interrupt Initialize function */

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

If the RC7 pin is pulled to GND for more than the Timer4 Period (500 ms), TMR4 will trigger TMR2 to stop and act as a one-shot Reset.

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

10. Using TMR4 as HLT to Generate an Interrupt

This example will present how to use the TMR4 as a Hardware Limit Timer (HLT) in order to generate an interrupt and stop TMR2 that also stops the ADCC auto-conversion.

This application will blink the LED0 with Timer2 Period (100 ms), if the potentiometer value is below a desired threshold, and will keep the LED in an ON state constantly if the potentiometer value is above that value. If the ADCC read value is above the maximum threshold and the RC7 pin is pulled to GND for more than Timer4 Period (500 ms), TMR4 will stop TMR2 and LED0 will blink with a 500 ms period for as long as RC7 is tied to GND.

One practical use for this code example is in a motor control application where the ADCC reads the shunt current at a fixed frequency. The user needs to compare that value with a maximum current and, if it is above for more than a period, then the user will stop the motor since it is consuming too much power.

This example uses the PIC18F47Q10 Curiosity Nano board with a POT Click, both inserted into a Curiosity Nano adapter.

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

- System clock initialization
- Port initialization
- PPS initialization
- ADCC initialization
- Timer2 initialization
- Timer4 initialization
- Interrupts handling and initialization

10.1 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 MCC from the toolbar (more information about how to install the MCC plug-in can be found here).
3. Go to *Project Resources → System → System Module* and make the following configurations:
   - Oscillator Select: HFINTOSC
   - HF Internal Clock: 1 MHz
   - Clock Divider: 1
   - In the Watchdog Timer Enable field in the **WWDT** tab, **WDT Disabled** has to be selected
   - In the **Programming** tab, **Low-Voltage Programming Enable** has to be checked
4. From the Device Resources window, add TMR2, TMR4, ADCC and do the following configurations:
   Timer2 Configuration:
   - Enable Timer: checked
   - Control Mode: Roll over pulse
   - Ext. Reset Source: TMR4_postscaled
   - Start/Reset Option: Starts at T2ON = `1` and TMR2_ers = `0`
   - **Timer Clock** tab
     - Clock Source: LFINTOSC
     - Clock Prescaler: 1:64
     - Postscaler: 1:1
   - Set 100 ms period in the **Timer Period** tab

   Timer4 Configuration:
   - Enable Timer: checked
   - Control Mode: Roll over pulse
   - Ext. Reset Source: T4INPPS

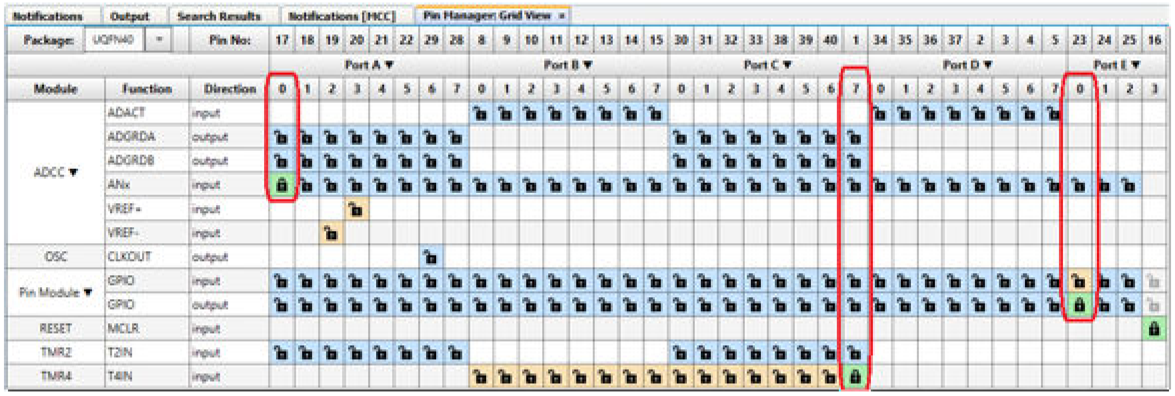
- Start/Reset Option: Resets at TMR4_ers = 1
- **Timer Clock** tab
  - Clock Source: LFINTOSC
  - Clock Prescaler: 1:64
  - Postscaler: 1:1
- Set 500 ms period in the **Timer Period** tab
- Enable Timer Interrupt: checked

ADCC Configuration:

- Enable ADC: Checked
- Operating: Basic mode
- • In the **ADC** tab, check the following options:
    - *ADC Clock → Clock Source*: Select **FRC**
    - Auto-conversion Trigger: Select **TMR2**
  - **CVD Features** tab:
    - Enable ADC Interrupt: checked

5. Open *Pin Manager → Grid View* window, select UQFN40 in the MCU package field, and select the I/O pins outputs to enable the internal signal access to the I/O.

Figure 10-1. Pin Mapping



Package:	UQFN40	Pin No:	17	18	19	20	21	22	29	28	8	9	10	11	12	13	14	15	30	31	32	33	38	39	40	1	34	35	36	37	2	3	4	5	23	24	25	16
			Port A ▼								Port B ▼								Port C ▼								Port D ▼								Port E ▼			
Module	Function	Direction	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3
ADCC ▼	ADACT	input									🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓									🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓				
	ADGRDA	output	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓									🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓												
	ADGRDB	output	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓									🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓												
	ANx	input	🔒	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	
	VREF+	input				🔓																																
	VREF-	input			🔓																																	
OSC	CLKOUT	output							🔓																													
Pin Module ▼	GPIO	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	GPIO	output	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔒	🔓	🔓	🔓
RESET	MCLR	input																																				🔒
TMR2	T2IN	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓									🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓												
TMR4	T4IN	input									🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔒												

6. Go to *Project Resources → Pin Module → RA0(ANA0)* and select only the **Analog** box. For the RC7 pin, select **WPU**, rename IO_RE0 to LED0 and select **Output** box .
7. Click **Generate** in the **Project Resources** tab.
8. Add these lines into the `main.c` file:

```
#define DesiredThreshold    300         /* Desired threshold value */
#define MaxThreshold        500         /* Maximum threshold value */
volatile uint16_t adcVal;

void TMR4_interrupt(void)
{
    /* HLT trigger: if adcVal > MaxThreshold and pin RC7 pulled-down */
    if (adcVal > MaxThreshold)
    {
        /* Toggle LED0 at the Timer2Period frequency */
        LED0_Toggle();
        /* HLT will stop TMR2 that also stops ADCC */
        TMR2_Stop();
    }
}
void ADCC_interrupt(void)
{
    /* This will toggle at a rate of 10Hz if adcVal < DesiredThreshold */
    if (adcVal < DesiredThreshold)
    {
        LED0_Toggle();
    }
    adcVal = ADCC_GetConversionResult();
}

void main(void)
{
    // Initialize the device
    SYSTEM_Initialize();
    TMR4_SetInterruptHandler(TMR4_interrupt);
    ADCC_SetADIInterruptHandler(ADCC_interrupt);
    // Enable the Global Interrupts
    INTERRUPT_GlobalInterruptEnable();
    // Enable the Peripheral Interrupts
    INTERRUPT_PeripheralInterruptEnable();

    while (1)
    {
        if ((adcVal > DesiredThreshold)&&(adcVal < MaxThreshold))
        {
            LED0_SetLow();
        }
    }
}
```

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

10.2 Bare Metal Code

The application with bare metal code will have the same behavior as the MCC generated code.

The first step will be to configure the microcontroller to disable the Watchdog Timer (WDT) and to enable Low-Voltage Programming (LVP).

```
#pragma config WDTE = OFF   /*disable Watchdog*/
#pragma config LVP = ON  /* Low voltage programming enabled, RE3 pin is MCLR */
```

Then, the following variables need to be defined:

```
#define Timer2Period        0x2F        /* TMR2 Period is 100ms */
#define Timer4Period        0xF1        /* TMR4 Period is 500ms */
#define DesiredThreshold    300         /* Desired threshold value */
#define MaxThreshold        500         /* Maximum threshold value */
#define AnalogChannel       0x00        /* Use ANA0 as input for ADCC */
volatile uint16_t adcVal;               /* ADCC global result value */
```

The `CLK_Initialize` function initializes the HFINTOSC internal oscillator:

```
static void CLK_Initialize(void)
{
    /* set HFINTOSC Oscillator */
    OSCCON1 = 0x60;
    /* set HFFRQ to 1 MHz */
    OSCFRQ = 0x00;
}
```

The `PPS_Initialize` function has the role to configure the RC7 peripheral select as input for TMR4:

```
static void PPS_Initialize(void)
{
    /* Set RC7 as input for TMR4 (T4IN) */
    T4INPPS = 0x17;
}
```

`PORT_Initialize` configures the RC7 digital input and RE0 output pins used for LED0:

```
static void PORT_Initialize(void)
{
    /* Set RC7 pin as digital */
    ANSELC = 0x7F;
    /* Set RE0 pin as output */
    TRISE = 0x06;
    /* Enable weak pull-up on pin RC7 */
    WPUC = 0x80;
}
```

`ADCC_Initialize` configures the TMR2 to be an auto-conversion trigger and enables the ADCC Interrupt flag:

```
static void ADCC_Initialize(void)
{
    /* ADACT Auto-Conversion Trigger Source is TMR2 */
    ADACT = 0x04;
    /* ADGO stop; ADFM right; ADON enabled; ADCONT disabled; ADCS FRC */
    ADCON0 = 0x94;
    /* Clear the ADCC interrupt flag */
    PIR1bits.ADIF = 0;
    /* Enabling ADCC interrupt flag */
    PIE1bits.ADIE = 1;
}
```

The `TMR2_Initialize` function sets the clock source and the registers needed to generate an 100 ms period:

```
static void TMR2_Initialize(void)
{
    /* TMR2 Clock source, LFINTOSC (00100) has 31 kHz */
    T2CLKCON = 0x04;
    /* T2PSYNC Not Synchronized; T2MODE Starts at T2ON = 1 and TMR2_ers = 0; T2CKPOL Rising 
Edge */
    T2HLT = 0x02;
    /* TMR2 external reset is TMR4_postscaled */
    T2RST = 0x02;
    /* TMR2 ON on; T2 CKPS Prescaler 1:64; T2 OUTPS Postscaler 1:1
       Minimum timer period is 31 kHz/64 = 2.064516 ms  */
    T2CON = 0xE0;
    /* Set TMR2 period, PR2 to 100ms */
    T2PR = Timer2Period;
    /* Clear the TMR2 interrupt flag */
    PIR4bits.TMR2IF = 0;
}
```

The `TMR4_Initialize` function sets the clock source and the registers needed to generate a 500 ms period:

```
static void TMR4_Initialize(void)
{
    /* TMR4 Clock source, LFINTOSC (00100) has 31 kHz */
    T4CLKCON = 0x04;
    /* T4PSYNC Synchronized; T4MODE Resets at TMR4_ers = 1; T4CKPOL Rising Edge */
    T4HLT = 0x87;
    /* TMR4 External reset signal by T4INPPS pin */
    T4RST = 0;
    /* TMR4 ON on; T4 CKPS Prescaler 1:64; T4 OUTPS Postscaler 1:1
       Minimum timer period is 31 kHz/64 = 2.064516 ms  */
    T4CON = 0xE0;
    /* Set TMR4 period, PR4 to 500ms */
    T4PR = Timer4Period;
    /* Clear the TMR4 interrupt flag */
    PIR4bits.TMR4IF = 0;
    /* Enabling TMR4 interrupt flag */
    PIE4bits.TMR4IE = 1;
}
```

The following initialization function will safely enable the Global and Peripherals interrupts, after all modules have been initialized with proper settings:

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

This function handles the two interrupts, checks the status of the Interrupt flag, and then calls the `TMR4_Interrupt` or `ADCC_Interrupt` functions:

```
static void __interrupt() INTERRUPT_manager (void)
{
    /* Interrupt handler */
    if (INTCONbits.PEIE == 1)
    {
        if (PIE4bits.TMR4IE == 1 && PIR4bits.TMR4IF == 1)
        {
            TMR4_Interrupt();
        }
        else if (PIE1bits.ADIE == 1 && PIR1bits.ADIF == 1)
        {
            ADCC_Interrupt();
        }
    }
}
```

The `ADCC_Interrupt` function first clears the Interrupt flag, toggles the LED0 (this will happen with Timer2 Period frequency), and then reads the ANA0 analog channel.

```
static void ADCC_Interrupt(void)
{
    /* Clear the ADCC interrupt flag */
    PIR1bits.ADIF = 0;

    if (adcVal < DesiredThreshold)
    {
        /* Toggle LED0 at the Timer2Period frequency */
         LATEbits.LATE0 = ~LATEbits.LATE0;
    }
    /* Get the conversion result from ADCC AnalogChannel */
    adcVal = ADCC_ReadValue(AnalogChannel);
}
```

The ADCC read function only needs a parameter (the channel that needs to be read):

```
static uint16_t ADCC_ReadValue(uint8_t channel)
{
    ADPCH = channel; /* Set the input channel for ADCC */
    /* TMR2 is trigger source for auto-conversion for ADCC */
    return ((uint16_t)((ADRESH << 8) + ADRESL));
}
```

The `TMR4_Interrupt` function first clears the Interrupt flag (if the ADCC read value is above the maximum threshold and if the RC7 pin is pulled to GND for more than 500 ms), TMR4 will stop TMR2 and LED0 will blink for a 500 ms period, as long as RC7 is tied to GND.

```
static void TMR4_Interrupt(void)
{
     /* Clear the TMR4 interrupt flag */
    PIR4bits.TMR4IF = 0;
    /* HLT trigger condition: if adcVal > MaxThreshold and pin RC7 is pulled-down */
    if (adcVal > MaxThreshold)
    {
        /* Toggle LED0 at the Timer4Period frequency */
        LATEbits.LATE0 = ~LATEbits.LATE0;
        /* HLT will stop TMR2 that also stops ADCC */
        /* Stop the Timer by writing to TMRxON bit */
        T2CONbits.TMR2ON = 0;
    }
}
```

The next code in the `void main` function, is an infinite loop (using a `while(1)`) to check for the ADCC value, using an “`if`” statement.

```
void main(void)
{
    /* Initialize the device */
    CLK_Initialize();             /* Oscillator Initialize function */
    PPS_Initialize();             /* Peripheral select Initialize function */
    PORT_Initialize();            /* Port Initialize function */
    ADCC_Initialize();            /* ADCC Initialize function */
    TMR2_Initialize();            /* TMR2 Initialize function */
    TMR4_Initialize();            /* TMR4 Initialize function */
    INTERRUPT_Initialize();       /* Interrupt Initialize function */

    while (1)
    {
        if ((adcVal > DesiredThreshold)&&(adcVal < MaxThreshold))
        {
            /* turn LED0 ON by writing pin RE0 to low */
            LATEbits.LATE0 = 0;
        }
    }
}
```

This will check if the read value from the Potentiometer (POT Click) is between two values. If so, the LED0 will turn on without blinking with a frequency. The ADCC interrupt is triggered by TMR2 to complete a conversion at a frequency determined by the Timer2 Period.

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

11. Using TMR2 as Alternate SPI Clock

The Timer 2 peripheral can be used as an alternative clock for the MSSP peripheral and can be used in the SPI clock configuration.

For example, to create a 10 kHz SPI clock, modify the Timer Period to 50 µs. This corresponds to 20 kHz since the SPI uses TMR2output/2 as clock, meaning that 20 kHz / 2 = 10 kHz frequency.

The following code examples will present how to set up the TMR2 peripheral and be used as clock source for the SPI configured as Master communicating to two Slave devices, alternatively.

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

- System clock initialization
- TMR2 initialization
- SPI1 initialization
- PPS initialization
- Port initialization
- Slave control functions
- Data exchange function

11.1 MCC Generated Code

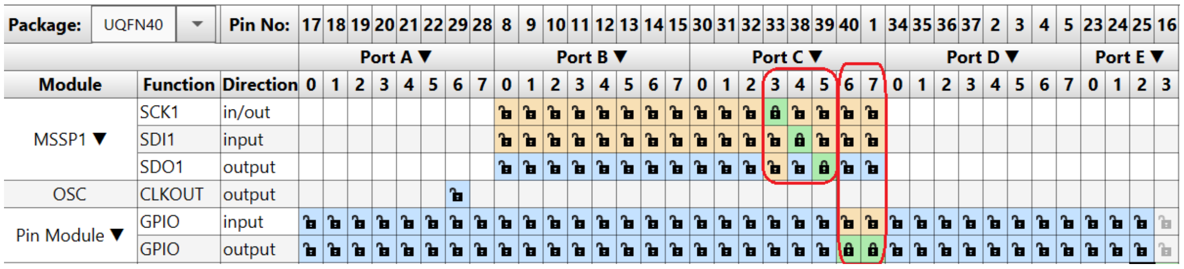
To generate this project using MPLAB Code Configurator (MCC) the next steps need to be followed:

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 make the following configurations:
   – Oscillator Select: HFINTOSC
   – HF Internal Clock: 4 MHz
   – Clock Divider: 1
   – In the Watchdog Timer Enable field in the **WWDT** tab, **WDT Disabled** has to be selected
   – In the **Programming** tab, **Low-Voltage Programming Enable** has to be checked
4. From the Device Resources window, add TMR2, MSSP1 and make the following configurations:
   Timer2 Configuration:
   – Enable Timer: checked
   – **Timer Clock** tab
      - Clock Source: HFINTOSC
      - Clock Prescaler: 1:64
      - Postscaler: 1:1
   – Set 50 µs period in the **Timer Period** tab

   MSSP1 Configuration:
   – Serial Protocol: SPI
   – Mode: Master
   – SPI Mode: SPI Mode 0
   – Input Data Sampled At: Middle
   – Clock Source Selection: TMR2/2
   – Actual Clock Frequency (Hz): 10000.00
5. Open *Pin Manager → Grid View* window, select UQFN40 in the MCU package field, and make the following pin configurations:
   – Set Port C pin 6 (RC6) as output for Slave Select 1 (SS 1)
   – Set Port C pin 7 (RC7) as output for Slave Select 2 (SS 2)

   The SCK, SDO and SDI pins appear alongside the MSSP1 peripheral and have their direction preset.

Figure 11-1. Pin Mapping



Package:	UQFN40	Pin No:	17	18	19	20	21	22	29	28	8	9	10	11	12	13	14	15	30	31	32	33	38	39	40	1	34	35	36	37	2	3	4	5	23	24	25	16
			Port A ▼								Port B ▼								Port C ▼								Port D ▼								Port E ▼			
Module	**Function**	**Direction**	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3
MSSP1 ▼	SCK1	in/out									🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔒	🔓	🔓	🔓	🔓												
	SDI1	input									🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔒	🔓	🔓	🔓												
	SDO1	output									🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔒	🔓	🔓												
OSC	CLKOUT	output							🔓																													
Pin Module ▼	GPIO	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	GPIO	output	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔒	🔒	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓

6. Click Pin Module in the Project Resources and set the custom names SS pins:
   – Rename RC6 to Slave1
   – Rename RC7 to Slave2
7. Click **Generate** in the **Project Resources** tab.
8. In the `main.c` file generated by MCC, add the following code:
   – Control of Slave devices
   – Data transmission

```
uint8_t writeData = 1;           /* Data that will be transmitted */
uint8_t receiveData;             /* Data that will be received */

void main(void)
{
    // Initialize the device
    SYSTEM_Initialize();

    while (1)
    {
        SPI1_Open(SPI1_DEFAULT);
        Slave1_SetLow();
        receiveData = SPI1_ExchangeByte(writeData);
        Slave1_SetHigh();
        SPI1_Close();

        SPI1_Open(SPI1_DEFAULT);
        Slave2_SetLow();
        receiveData = SPI1_ExchangeByte(writeData);
        Slave2_SetHigh();
        SPI1_Close();
    }
}
```

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

11.2 Bare Metal Code

The first step will be to configure the microcontroller to disable the Watchdog Timer and to enable Low-Voltage Programming (LVP).

```
/* WDT operating mode->WDT Disabled */
#pragma config WDTE = OFF
/* Low voltage programming enabled, RE3 pin is MCLR */
#pragma config LVP = ON
```

The internal oscillator has to be set to the desired value. This example uses the HFINTOSC with a frequency of 4 MHz. This translates in the following function:

```
static void CLK_init(void)
{
    OSCCON1 = 0x60;              /* set HFINTOSC Oscillator */
    OSCFRQ  = 0x02;              /* set HFFRQ to 4 MHz */
}
```

The following function initializes the Timer2 peripheral with the HFINTOSC clock:

```
static void TMR2_Initialize(void)
{
    /* TMR2 Clock source, HFINTOSC (00011) */
    T2CLKCON = 0x03;
    /* T2PSYNC Not Synchronized, T2MODE Software control, T2CKPOL Rising Edge */
    T2HLT = 0x00;
    /* TMR2ON on; T2CKPS Prescaler 1:1; T2OUTPS Postscaler 1:1 */
    T2CON = 0x80;
    /* Set TMR2 period, PR2 to 199 (50us) */
    T2PR = Timer2Period;
    /* Clear the TMR2 interrupt flag */
    PIR4bits.TMR2IF = 0;
}
```

The `SPI1_Initialize` function will configure the SPI clock source to be TMR2 Output/2:

```
static void SPI1_Initialize(void)
{
    /* SSP1ADD = 1 */
    SSP1ADD = 0x01;
    /* Enable module, SPI Master Mode, TMR2 as clock source */
    SSP1CON1 = 0x23;
}
```

Therefore, the SPI pins can be relocated using the SSPxCLKPPS, SSPxDATPPS, SSPxSSPPS registers for the input channels and by using the RxyPPS registers for output channels.

The method to configure the location of the pins is independent of the application purpose and the SPI mode. Each microcontroller has its own default physical pin position for peripherals, but they can be changed using the Peripheral Pin Select (PPS).

For SPI1 in Master mode, only the SDI pin needs to be input so it is used with its default location RC4. SCK was mapped to RC3 and SDO was mapped to RC5. This translates into the following code:

```
static void PPS_Initialize(void)
{
    RC3PPS = 0x0F;               /* SCK channel on RC3 */
    SSP1DATPPS = 0x14;           /* SDI channel on RC4 */
    RC5PPS = 0x10;               /* SDO channel on RC5 */
}
```

Since this example has the Master sending data to two Slave devices, two SS pins are needed (SS1 and SS2). For both, a General Purpose Input/Output (GPIO) pin was used (RC6 for SS1 and RC7 for SS2).

Table 11-1. SPI Pin Locations

Channel	Pin
SCK	RC3
SDI	RC4
SDO	RC5
SS1	RC6
SS2	RC7

Since the Master devices control and initiate transmissions, the SDO, SCK and SS pins must be configured as output while the SDI channel will keep its default direction as input. The following example is based on the relocation of the SPI1 pins made above:

```
static void PORT_Initialize(void)
{
    ANSELC = 0x07;      /* Set RC6 and RC7 pins as digital */
    TRISC  = 0x17;      /* Set SCK, SDO, SS1, SS2 as output and SDI as input */
}
```

A Master will control a Slave by pulling low the SS pin. If the Slave has set the direction of its SDO pin to output (when the SS pin is low), the SPI driver of the Slave will take control of the SDI pin of the Master, shifting data out from its Transmit Buffer register.

All Slave devices can receive a message, but only those with the SS pin pulled low can send data back. It is not recommended to enable more than one Slave in a typical connection since all of them will try to respond to the message and the Master has only one SDI channel. Therefore, the transmission will result in a write collision.

Before sending data, the user must pull low one of the configured SS signals to let the correspondent Slave device know it is the recipient of the message.

```
static void SPI1_slave1Select(void)
{
    LATCbits.LATC6 = 0;          /* Set SS1 pin value to LOW */
}
```

Once the user writes new data into the Buffer register, the hardware starts a new transfer, generating the clock on the line and shifting out the bits. The bits are shifted out starting with the Most Significant bit (MSb).

When the hardware finishes shifting all the bits, it sets the Buffer Full Status bit. The user must check the state of the flag before writing new data into the register by constantly reading the value of the bit (or polling), or else a write collision will occur.

```
static uint8_t SPI1_exchangeByte(uint8_t data)
{
    SSP1BUF = data;

    while(!PIR3bits.SSP1IF) /* Wait until data is exchanged */
    {
        ;
    }
    PIR3bits.SSP1IF = 0;

    return SSP1BUF;
}
```

The user can pull the SS channel high if there is nothing left to transmit.

```
static void SPI1_slave1Deselect(void)
{
    LATCbits.LATC6 = 1;          /* Set SS1 pin value to HIGH */
}
```

The following function is the `int_main(void)` and begins peripheral initialization before the SPI commands are run in a infinite loop `while(1)`:

```
int main(void)
{
    CLK_Initialize();
    PPS_Initialize();
    PORT_Initialize();
    TMR2_Initialize();
    SPI1_Initialize();

    while(1)
    {
        SPI1_slave1Select();
        receiveData = SPI1_exchangeByte(writeData);
        SPI1_slave1Deselect();
        SPI1_slave2Select();
        receiveData = SPI1_exchangeByte(writeData);
        SPI1_slave2Deselect();
    }
}
```

View the PIC18F47Q10 Code Example on GitHub
Click to browse repositories

12. References

1. MPLAB Code Configurator User’s Guide
2. Getting Started with Writing C-Code for PIC16 and PIC18 Tech Brief

13. Revision History

Document Revision	Date	Comments
A	05/2020	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 specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "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

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE. Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights unless otherwise stated.

Trademarks

The Microchip name and logo, the Microchip logo, Adaptec, AnyRate, AVR, AVR logo, AVR Freaks, BesTime, BitCloud, chipKIT, chipKIT logo, CryptoMemory, CryptoRF, dsPIC, FlashFlex, flexPWR, HELDO, IGLOO, JukeBlox, KeeLoq, Kleer, LANCheck, LinkMD, maXStylus, maXTouch, MediaLB, megaAVR, Microsemi, Microsemi logo, MOST, MOST logo, MPLAB, OptoLyzer, PackeTime, PIC, picoPower, PICSTART, PIC32 logo, PolarFire, Prochip Designer, QTouch, SAM-BA, SenGenuity, SpyNIC, SST, SST Logo, SuperFlash, Symmetricom, SyncServer, Tachyon, TempTrackr, TimeSource, tinyAVR, UNI/O, Vectron, and XMEGA are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

APT, ClockWorks, The Embedded Control Solutions Company, EtherSynch, FlashTec, Hyper Speed Control, HyperLight Load, IntelliMOS, Libero, motorBench, mTouch, Powermite 3, Precision Edge, ProASIC, ProASIC Plus, ProASIC Plus logo, Quiet-Wire, SmartFusion, SyncWorld, Temux, TimeCesium, TimeHub, TimePictra, TimeProvider, Vite, WinPath, and ZL are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Adjacent Key Suppression, AKS, Analog-for-the-Digital Age, Any Capacitor, AnyIn, AnyOut, BlueSky, BodyCom, CodeGuard, CryptoAuthentication, CryptoAutomotive, CryptoCompanion, CryptoController, dsPICDEM, dsPICDEM.net, Dynamic Average Matching, DAM, ECAN, EtherGREEN, In-Circuit Serial Programming, ICSP, INICnet, Inter-Chip Connectivity, JitterBlocker, KleerNet, KleerNet logo, memBrain, Mindi, MiWi, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, MultiTRAK, NetDetach, Omniscient Code Generation, PICDEM, PICDEM.net, PICkit, PICtail, PowerSmart, PureSilicon, QMatrix, REAL ICE, Ripple Blocker, SAM-ICE, Serial Quad I/O, SMART-I.S., SQI, SuperSwitcher, SuperSwitcher II, Total Endurance, TSHARC, USBCheck, VariSense, ViewSpan, WiperLock, Wireless DNA, and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

The Adaptec logo, Frequency on Demand, Silicon Storage Technology, and Symmcom are registered trademarks of Microchip Technology Inc. in other countries.

GestIC is a registered trademark of Microchip Technology Germany II GmbH & Co. KG, a subsidiary of Microchip Technology Inc., in other countries.

All other trademarks mentioned herein are property of their respective companies.

© 2020, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

ISBN: 978-1-5224-6546-1

Quality Management System

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

Worldwide Sales and Service

AMERICAS

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

ASIA/PACIFIC

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

ASIA/PACIFIC

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

EUROPE

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