June 26, 2021

Kodiak North

Contents

[Hardware Used Throughout this Document 1](#_Toc78454006)

[Debug Mode 1](#_Toc78454007)

[Software PLIB Remarks 2](#_Toc78454008)

[UART Blocking 2](#_Toc78454009)

[UART Non-Blocking 4](#_Toc78454010)

[SPI 7](#_Toc78454011)

[CORE TIMER Periodic Interrupt Mode 9](#_Toc78454012)

[Rotary Encoders via Change Notice 10](#_Toc78454013)

[ADC 11](#_Toc78454014)

[Git and MPLAB 14](#_Toc78454015)

# Hardware Used Throughout this Document

* Olimex PIC-KIT3 programmer/debugger
* Olimex PIC32-HMZ144
  + Utilizes the PIZ32MZ2048EFM144 microprocessor

# Debug Mode

The Olimex PIC-KIT3 is a powerful device that not only allows us to program our microprocessor, but also debug it using MPLAB-X’s debug interface. Though not always necessary, a real debugger (rather than mere print statements) can greatly increase a developer’s ability to quickly find and resolve bugs in the system. However, before we can run the processor in debug mode, we must properly configure the PIC32MZ’s registers; namely DEVCFG0 and DEVCFG2. Fortunately, the MHC makes it easy to do this. Start by clicking on the ‘System’ block and navigating to Device & Project Configuration -> PIC32MZxxxxEFxxxx Device Configuration.

DEVCFG0:

First, expand the DEVCFG0 tab and set Background Debugger Enable bits (DEBUG) to ON. This sets a bit in the compiled hex file to allow for debugging. Also, the PIC-KIT3 and PIC32MZ are mapped to use ICSP2 for programming/debugging. Therefore, set the In-Circuit Emulator/Debugger Communication Channel Select bits (ICESEL) to ICS\_PGx2.

DEVCFG2:

The last thing to do is ensure the oscillator settings are correct for the board to run in debug mode. The board’s VCO must output from 350 MHz to 700 MHz. The default configuration through MPLAB-X and MCH is within this range, FVco = 400MHz, but it is good to know where this comes from and how to modify it. A block diagram of how FVco is calculated can be found in Figure 8-1 in the PIC32MZ Embedded Connectivity (EC) Family Data Sheet. The equation is as follows:

FVco = (FPLLICLK / FPLLIDIV) \* FPLLMULT

By default, FPLLICLK is set to FRC (8MHz), FPLLIDIV is 1, and FPLLMULT is 50; equating to a FVco of 400MHz. These settings are all modifiable within the DEVCFG2 register.

The microprocessor is now properly configured to be run in debug mode. Set a breakpoint and try it out!

# Software PLIB Remarks

* When the main function is entered, be sure to call the function, SYS\_Initialize(NULL). This initializes all peripherals that are configured and generated with the Harmony 3 configurator. Be wary that the SYS\_Initialize function may not start all peripherals. For example, the function calls CORETIMER\_Initialize, however it does not call CORETIMER\_Start.
* Enable, disable, and restore global interrupts by using functions defined in plib\_evic.h/c

# UART Blocking

Test description:

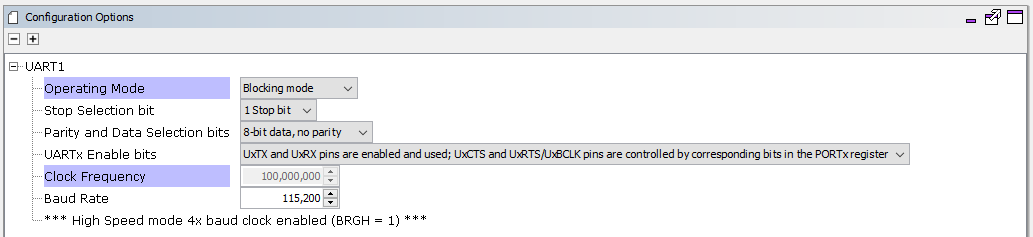
The goal is to configure a UART peripheral to send and receive data from an Arduino. There are three available modes of operation: blocking, non-blocking, and ring buffer. For this test, only the blocking mode is explored. Since the PIC32-HMZ144 development board lacks a serial COM port for logging data to a serial terminal, this UART peripheral will be used to send ‘printf’ messages to the Arduino for the Arduino to then display on a serial port. The Arduino will respond with an ACK to indicate that it has received the message. If no ACK is received, or the ACK does not match the expected value, an error occurred during transmission and an LED is illuminated.

Pin definitions:

The UART1 module is chosen arbitrarily. All UART peripheral pins are PPS, so there are many pins available for use. The following pins are chosen because they are 5V tolerant to ease the wiring to an Arduino:

|  |  |  |
| --- | --- | --- |
| UART Purpose | Pin Number | Pin ID |
| U1TX | 124 | RF0 |
| U1RX | 125 | RF1 |

PIC32MZ UART1 Configuration:

****

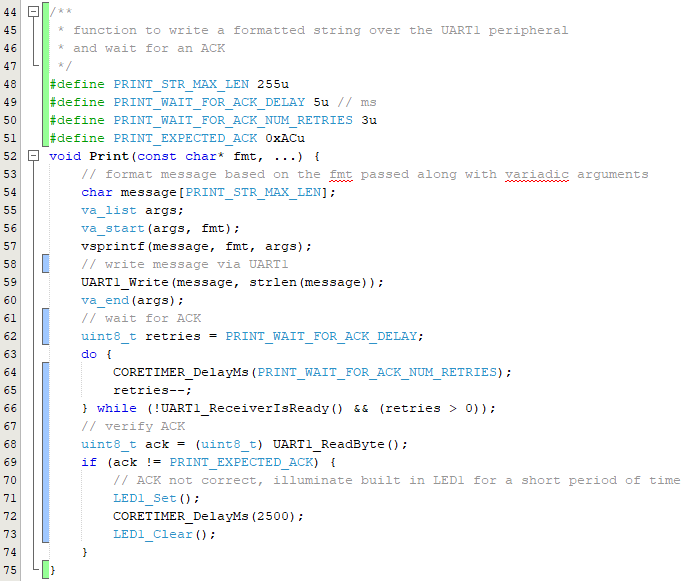
Arduino Serial1 (UART1) Configuration:

Default is 1 stop bit, 8-bit data, and no parity.

Results:

The following functions are investigated in this experiment. No odd behavior is found. See the code snippet below for an example of using a subset of these functions. In the end, using UART1 to serve as a ‘printf’ solution for the PIC32 works flawlessly.

* UART1\_Write() - transmits a buffer based on the size passed
* UART1\_Read() - reads into a buffer based on the size passed
* UART1\_ReadByte() - reads and returns one byte
* UART1\_ReceiverIsReady() - returns true if data is available in the UART1 receive FIFO
  + This function significantly helped the timing of communication when waiting to receive an ACK from the Arduino while the SPI is also sending/receiving data to/from the Arduino. Without it, serial log messages would fall out of order.
* UART1\_WriteByte() - transmits one byte
* UART1\_TransmitterIsReady() - returns false if the UART1 transmit FIFO is full



# UART Non-Blocking

Test Description:

Configure a UART peripheral in non-blocking mode to send and receive data from an Arduino. Implement a similar print-with-ack function with the UART in this mode.

Notes:

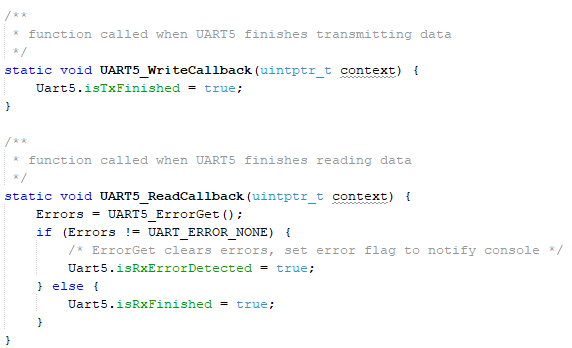
* The transmit and receive buffers are 9 bits wide (only 8 bits are used in 8-bit data mode) and 8 levels deep
* Non-blocking mode uses interrupts to handle the transmission and reception of data. When configured in MHC to be in non-blocking mode, all interrupt sources are enabled (Tx, Rx, Fault).
* Tx interrupt configured by MHC to fire whenever the transmit buffer is empty
  + The UTXISEL control bits (UxSTA<15:14>) are set to 0b10
* Rx interrupt configured by MHC to fire whenever the receive buffer is NOT empty
* Fault interrupt unconfigurable. When enabled, fault interrupt fires whenever a Parity Error, Framing Error, or Receive Buffer Overflow occurs

Pin definitions:

Refer to the section above as the same pins are used.

Results:

UART5 is used for this experiment. With the peripheral pin select functionality, the exact same pins for UART1 Tx/Rx are reused. Now that the peripheral is configured for non-blocking mode, care should be taken to not delay the processor, for instance, by using the CORETIMER\_DelayMs function, or by adding other blocking code. Each available UART5 interrupt handler should have a callback attached where the programmer sets a flag to indicate an operation has completed. These flags will be sampled in a main loop and actions will be taken when necessary. Using the UART5\_WriteCallbackRegister and UART5\_ReadCallbackRegister functions, attach the following callbacks to the corresponding register. Once attached, the write callback will fire when data transmission is complete, and the read callback will fire when data reception is complete or an error is detected.



Note that a struct, Uart5, is used to contain the three boolean flags, isTxFinished, isRxErrorDetected, and isRxFinished. Init isTxFinished to false, isRxErrorDetected to false, and isRxFinished to true. The latter being set to true may not make sense now, but it will be explained in detail later.

These three flags should be sampled for true in the program’s main loop. Check for errors first, else check for Rx complete, else check for Tx complete. If a flag is set, immediately clear it within the conditional statement, and perform an operation. For the sake of simplicity, if an error is detected, notify the developer by illuminating an LED. When the system is working properly, parity, framing, and overrun errors will not occur frequently. If Rx is finished, the program has either just been initialized, or the Arduino has sent an ACK. It is now a good time to send data - either a start byte or a message to print. Once transmission is complete, the isTxFinished flag will be set as an indicator, and we know that the Arduino should respond with an ACK. Therefore it is a good idea to initiate a read of one byte. Note that this read is non-blocking. The system indicates that the read is complete with the isRxFinished flag, and we repeat from the top. See the following pseudo code:

*while 1*

*if isRxErrorDetected*

*illuminate and LED*

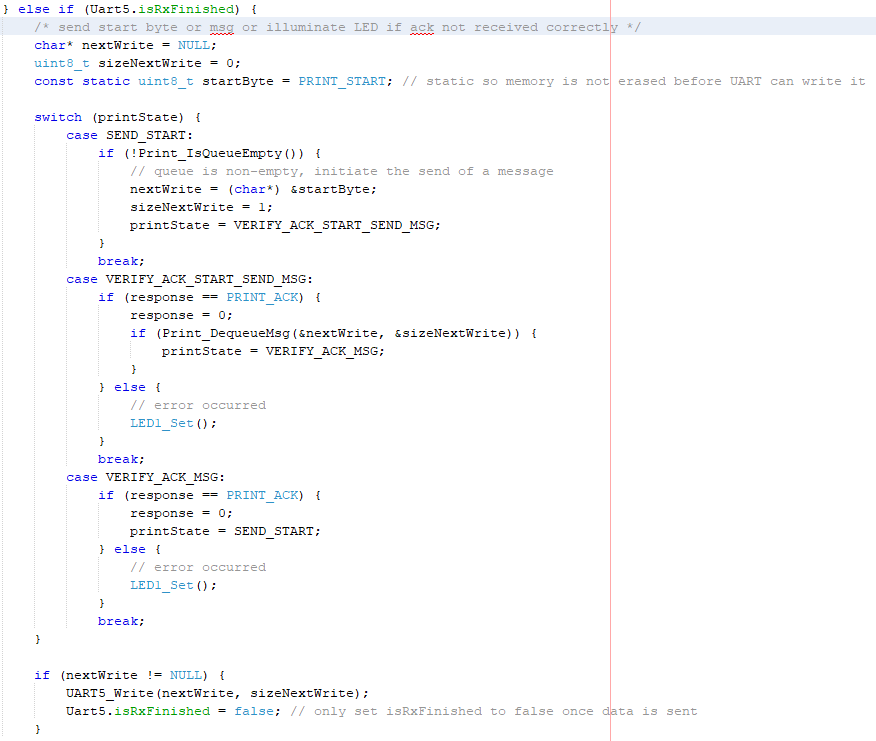
*else if isRxFinished*

*send data*

*else if isTxFinished*

*initiate read of one byte*

This pseudo code will do the trick, however for added robustness, a state machine is implemented within the isRxFinished condition to flip flop between sending start bytes, verifying ACKs, and sending messages. See the figure below. Consequently, the Arduino is configured to wait for a start byte, send an ACK upon receiving, wait for a message, and then send another ACK upon receiving. With all of this done, the UART5 peripheral can be used to send printf style messages to an Arduino in a non-blocking fashion.



# SPI

Test description:

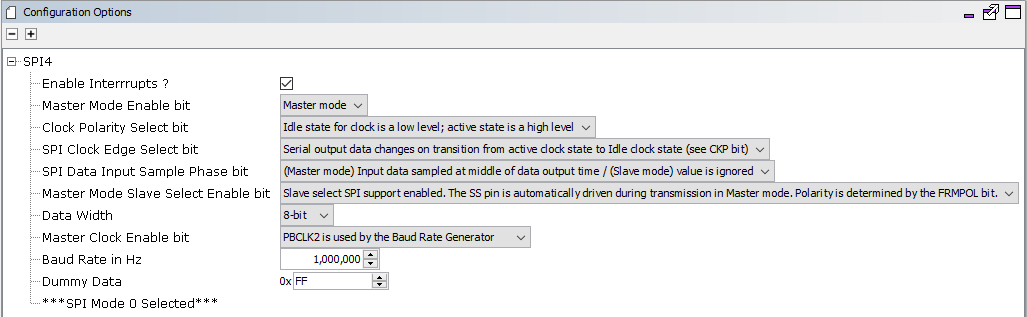
The goal is to configure one of the PIC32MZ’s SPI modules to communicate with an Arduino via 8-bit data width. Since Arduino’s are 8-bit microcontrollers, 8-bit data width is the only option. Various write/read functions will be explored to gain an understanding on when to use which function. The PIC32 will serve as the SPI master, therefore the Arduino will be the SPI slave.

Pin definitions:

The SPI4 module is chosen as SCK4 (pin 98, RD10) is available within the pin configuration provided from Shoab/Jaffer. All other pins are PPS (Peripheral Pin Select), meaning that there are multiple pins on the chip that can be selected as SS4/SDO4/SDI4. The developer choses which pins to use with the Harmony 3 Configurator. The following pins are selected for use. These pins are all 5V tolerant which removes the need for any TTL level manipulation to communicate with the 5V Arduino.

|  |  |  |
| --- | --- | --- |
| SPI Purpose | Pin Number | Pin ID |
| SS4 (out) | 97 | RD9 |
| SCK4 | 98 | RD10 |
| SDI4 (MISO) | 99 | RD11 |
| SDO4 (MOSI) | 104 | RD0 |

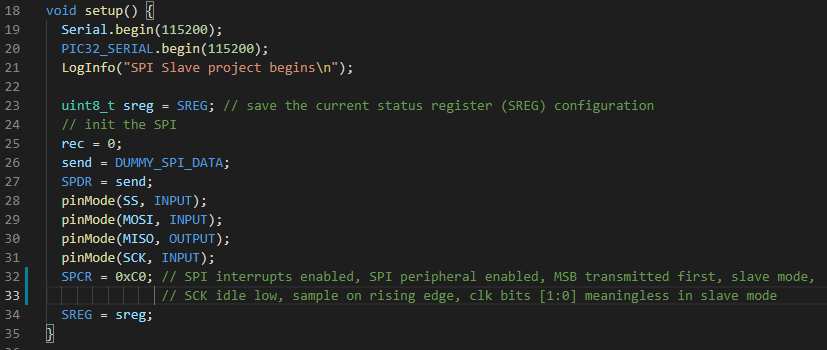
PIC32MZ SPI4 Master Configuration:

****

Note that though interrupts are enabled, they are unused throughout this experiment. Also note that some SPI settings are not modifiable from within the Harmony 3 configurator such as:

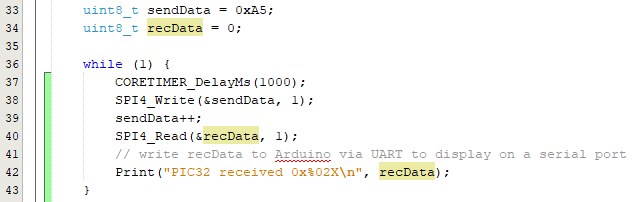
* PIC32MZ transmits MSB first
* SS driven LOW when active

Arduino SPI Slave Configuration:

****

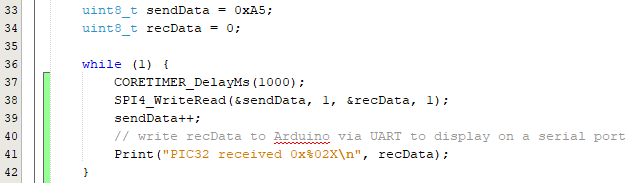
Results:

The functions SPI4\_WriteRead, SPI4\_Write, and SPI4\_Read are tested. Though use of the SPI is straightforward, there is unexpected behavior with the SPI4\_Read function. The PIC32 is configured to write a byte to the Arduino, increment the data, then read the Arduino’s response every second. See the following code block. The Arduino is configured to respond with 0xFF whenever data is received.



With this implementation, the Arduino always receives the sendData byte, however recData is always 0x00. I think this is because SPI is full duplex. Since the software is not prepared to read as it is writing, the Arduino’s response is missed. Perhaps the use of interrupts could mitigate this problem.

Fortunately, there is a library function to accommodate this, SPI4\_WriteRead. By using this function, the PIC32 receives the expected 0xFF response. Therefore, it is apparent that the SPIx\_WriteRead function should be used when reading data back from the slave. Otherwise, the response will be missed. See the following code block.



Follow up:

After revisiting this code and experimenting with interrupts enabled or disabled, a few noteworthy points have been discovered. If the SPI does not have interrupts enabled, it will block while sending/receiving data. If a developer would not want this code to block, interrupts must be enabled. Even with interrupts enabled, it will be easiest to use the SPIx\_WriteRead function to read data from the slave device while transmitting. However, since the code is non-blocking with interrupts enabled, it should be possible to call SPIx\_Write and then SPIx\_Read right after to capture data sent from the slave. Though this is not recommended because the SPIx\_WriteRead function will do it all with one call.

With interrupts enabled, a callback can be attached to the SPI module. The exact same callback is fired whenever the transmission **or** reception of data completes unless the developer sets a new callback during either of these operations. In the Harmony SPI Interrupt example project, the callback is simply used to set a flag that indicates the completion of a data transfer. More useful callbacks can be implemented, but the use of a callback is not necessary with the SPI in interrupt mode. Development can easily be done with or without a SPI callback. When compared with the UART peripheral in interrupt mode, though the use of UART callbacks is also not necessary, they significantly improve the robustness of a developed UART driver.

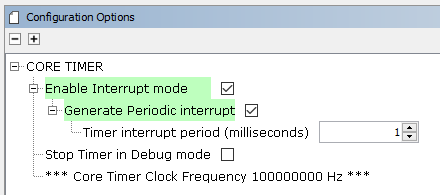
# CORE TIMER Periodic Interrupt Mode

Test Description:

Configure the CORE TIMER peripheral to interrupt at a frequency of 1kHz to track time at millisecond intervals.

Results:

CORE TIMER is set up with the Harmony 3 configurator as per the figure below. Then the code is generated.



Though Harmony 3 generates a nice API, that even contains a tickCounter variable, it provides no means of accessing the tickCounter unless the developer declares it as external elsewhere in the program. A neat way to track time is by using the API’s *CORETIMER\_CallbackSet* function. The attached callback is called every time the CORE TIMER module triggers its periodic interrupt. The process is as follows:

1. Create a global variable to track time
   1. *volatile uint32\_t msTicks = 0;*
      1. Note that this variable must be declared as **volatile** since it is accessed from within an interrupt as well as elsewhere in the code.
2. Create the callback function with the following signature
   1. *void CORETIMER\_InterruptCallback(uint32\_t status, uintptr\_t context);*
3. Within the function implementation, simply increment the value of *msTicks.*
4. In the project’s main function add the following lines of code:
   1. *SYS\_Initialize(NULL);*
   2. *CORETIMER\_CallbackSet(CORETIMER\_InterruptCallback, (uintptr\_t) NULL);*
   3. *CORETIMER\_Start();*
5. Now access *msTicks* anywhere it is needed in the software.
   1. Note that if strange errors occur when reading from *msTicks*, it may be necessary to disable interrupts and re-enable them after reading. However, no strange errors are noticed in this simple test project.

# Rotary Encoders via Change Notice

Test description:

Configure a change notice interrupt to sample the A and B phases off a rotary encoder. The interrupt will track revolutions of the encoder for use elsewhere in the code.

Notes:

* Every I/O port pin (RAx-RKx) can be used as a change notification pin (CNAx-CNKx).
* XC32 vector names: \_CHANGE\_NOTICE\_x\_VECTOR where x = A-K
* Control registers: CNENx, CNNEx, CNSTATx, CNFx, CNPUx, CNPDx, CNCONx
  + EDGEDETECT bit (CNCONx<11>) plays an important role
  + Pull-ups and pull-downs on change notification pins should always be **disabled** when the port pin is configured as a digital output
* Change noticed is enabled for an entire port, not for individual pins on the port

Pin definitions:

The E2 encoders, used by MMP for their motors, operate with 5V TTL. Therefore, the following pins are chosen because they are 5V tolerant.

* Encoder A phase - RJ13
* Encoder B phase - RJ14

Pin configuration:

Two pins on PORTJ will be used. To enable change notification on these pins, a couple boxes must be checked in various places within the Harmony 3 Configurator. First, navigate to Tools->EVIC Configuration, scroll down to vector number 126, and check off the ‘Use’ box next to CHANGE\_NOTICE\_J (PORTJ Input Change). Next, navigate to tools->Pin Configuration and find pin number 28 and 29 (RJ13 and RJ14). Set these pins as GPIO, In direction, and check off the ‘Change Notification (CNEN)’ box for both pins. Depending on the specific encoder hardware, it may be useful to enable a pull-up or pull-down resistor on the pins. Therefore, if necessary, check off the desired box when enabling Change Notification in the Pin Configuration window. These pins are now configured properly for change notification, and it is time to generate the new code.

Results:

With the pins configured and code generated, a couple files will be modified. Firstly, an interrupt vector for change notice on PORTJ is added to the file interrupts.c. Secondly, the file plib\_gpio.c now implements the CHANGE\_NOTICE\_J\_InterruptHandler function, and it provides the function GPIO\_PinInterruptCallbackRegister. This callback register function is used to attach a callback to pins RJ13 and RJ14. The callback will increment/decrement a global counter variable that tracks the encoder’s ticks. There is one final step before code can be tested - enabling the Change Notice interrupts in software upon initialization. This can be done with either of the two following lines of code:

* *<pin\_name>\_InterruptEnable();*
* *GPIO\_PinInterruptEnable(<pin\_name>\_PIN);*

Where *pin\_name* is the name of the pin assigned by the developer in the Harmony 3 Configurator. Now a rotation of the encoder will cause the attached callback to fire and track the count of the encoder!

# ADC

Test description:

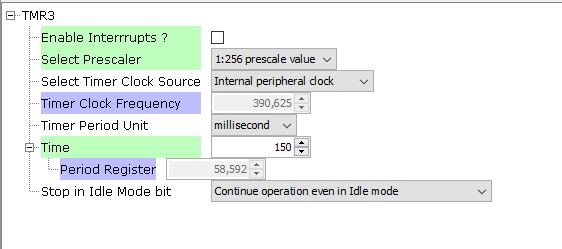
Configure the ADC in interrupt mode and toggle an LED when it reaches above a predetermined threshold. The example listed [here](https://microchip-mplab-harmony.github.io/csp_apps_pic32mz_ef/apps/adchs/adchs_interrupt/readme.html) is followed.

Pin configuration:

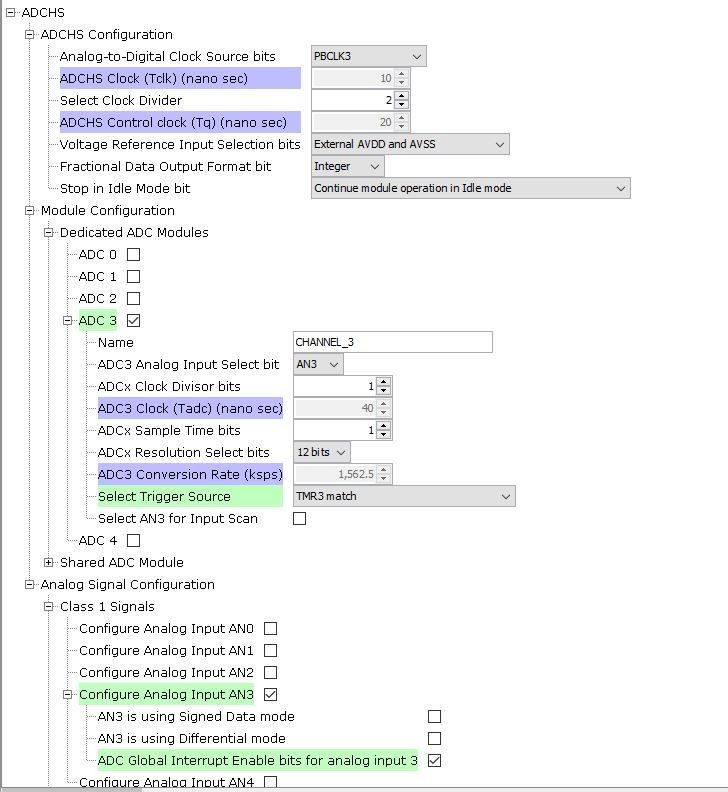
Port RB3 is used. Navigate to the pin configuration window within the Harmony 3 Configurator and set RB3’s function to AN3. RB3 is not 5V tolerant so be sure that the input device supplies no more than 3.3V.

Results:

The ADCHS peripheral requires a hardware timer to control its sample rate. Timer 3 is used and configured to trigger a match every 150ms. See the configuration below.



Now set up the ADCHS unit. For this example, ADC3 is used. Be sure to follow these settings closely as a small discrepancy could cause the whole module to break!



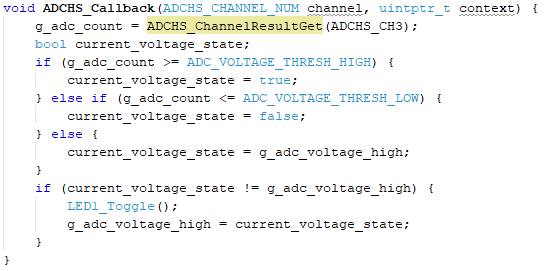
After all settings are applied, generate the code. Attention should now be directed to main.c.

Start off by creating a callback function to attach to ADC3.

*void ADCHS\_Callback(ADCHS\_CHANNEL\_NUM channel, uintptr\_t context);*

In interrupt mode, a callback **must** be used. Within the callback, the function *ADCHS\_ChannelResultGet(<channel>)* **must** be called, or else the processor will hang up.

The callback in this example is also populated with code to toggle the builtin LED if the ADC output is above a threshold. See the implementation below.



Lastly, attach the callback to ADC3, start Timer 3, flash the microcontroller, and watch the LED toggle.

*ADCHS\_CallbackRegister(ADCHS\_CH3, ADCHS\_Callback, (uintptr\_t) NULL);*

*TMR3\_Start();*

# Output Compare for PWM

Test Description:

Configure an output compare module to generate a PWM wave and vary its duty cycle with a potentiometer. Use an oscilloscope to verify that it is working properly.

Notes:

* Output compare Special Function Registers (SFRs)
  + OCxCON: Output Compare ‘x’ Control Register
  + OCxR: Output Compare ‘x’ Compare Register
  + OCxRS: Output Compare ‘x’ Secondary Compare Register
* PWM Mode
  + Developer can choose to include a Fault Protection Input
  + OCxR register is a read-only slave duty cycle register
  + OCxRS is a buffer register that is written to by the user to update the PWM duty cycle
  + On each timer to period register match event (end of the PWM period), OCxR is loaded with the contents of OCxRS
    - OCxR = 0x0000 -> 0% duty cycle
    - OCxR > PR -> 100% duty cycle
    - OCxR = PR -> OCx pin low for one time base count value and high for all other count values
  + PWM Period = [(PR + 1) • TPB • (TMR Prescale Value)]
    - PR: Period Register of the timer selected to run the output compare module
    - TPB: Time (period) of Peripheral Bus clock that is mapped to the output compare module
      * On the selected hardware, this is PB3 with a default frequency of 100MHz. Therefore the period is 1/100MHz = 10ns.
  + The PWM period must not exceed the Period Register of the selected timer. If the calculated period is too large, select a larger prescaler to prevent overflow.
  + To maintain maximum PWM resolution, select the smallest prescaler that does not result in an overflow
  + Maximum PWM Resolution (bits) = log10(PR + 1) / log10(2)

= log10(FPB / (FPWM • Prescaler)) / log10(2)

* + - To calculate, select desired PWM frequency and convert to period. Use PWM Period equation above to solve for PR. Plug in PR to this equation.

Pin configuration:

Results:

# Git and MPLAB

There are a few precautions a developer should take when using Git to version control a project with colleagues. Firstly, there are some generated files within the project’s root directory (.X folder) that should not be tracked by Git. Find a list of what to track and what not to track [here](https://microchipdeveloper.com/faq:72). If using the Harmony 3 tool, all of its generated files **should** be tracked, even though most are generated.

Before pushing:

* If using Harmony 3, document any changes to pin mappings and notify other developers. Harmony will complain the next time code is generated after a colleague pulls the code, but it is safe for the colleague to proceed as long as he/she is aware that these changes were purposely made.

Before pulling:

* Close the MPLAB-X IDE and all related tools.
* Use VS Code to resolve merge conflicts **WITH MPLAB-X CLOSED.**

After pulling:

* If using Harmony 3, launch the tool and regenerate code immediately.