

ENCM 369 PIC Activity 7

ENCM 369 ILS Winter 2021 Version 1.0

Activity Objectives

- Gain familiarity with the general concept of interrupts and interrupt handlers
- Learn how the PIC18 interrupt system works
- Apply interrupts to improve the 1ms system tick
- Make an interrupt-driven TimeXus function
- Practice following coding requirements and code review of a peer

Deliverables

- Interrupt-driven “Twinkle, twinkle, little star” song
- Check-in of your code to Github
- Completed rubric review of someone’s code in your learning community

Hardware Design

There is no new hardware on your PIC breadboard required for this activity. You should keep the configuration from PIC Activity 6 with RA2 as the DAC output. Connect this circuit into the input of your audio amplifier and filtered speaker circuit.

Background Information

When working with embedded systems, you have the fun of doing a lot “low level” programming which means interacting with the hardware and peripherals of the processor directly. This is typically something you never see when programming high level code on a PC, since there is an operating system, a ton of hardware drivers, and the BIOS (basic input output system) code already in place.

Perhaps the most powerful and important low-level feature to master is that of something called “interrupts.” The nice thing about interrupts is that the name defines exactly the function. Think about doing something like playing a video game, when suddenly your phone beeps with a text message and interrupts you. You pause the game, grab your phone to quickly respond, then resume the game exactly where you left off. The game has no idea that you paused it, but you were able to “service” the urgent need of your phone. Congratulations, you are an interrupt handler!

The concept of interrupts on an embedded processor is identical. Processors will happily execute line after line of code continuously until the power goes off. You have seen this happening in the PIC activities so far, where `main()` runs endlessly in the `while(1)` loop. Various functions are called to do their thing, but everything runs in a linear process. However, we’ve seen that doesn’t always happen quickly enough, and various delays can cause bigger timing issues. As systems become more complicated with more and more applications / tasks / functions running, timing can get even more disrupted. There are certain functions or events

that need to be handled immediately and/or with precise timing, which is where interrupts come in.

An interrupt is just a hardware signal to the processor that causes the processor to stop whatever it is doing, jump (aka “vector”) to a special function called an interrupt service routine (ISR), run that code, then resume running the main program as if nothing happened. You’ve already seen the hardware signals that trigger interrupts – the TMR0IF bit is the “Timer 0 Interrupt Flag” which is set by the timer peripheral when the timer expires. What we have not done yet is enable the interrupt system to respond automatically to this flag. That’s the purpose of this activity.

PIC18 Interrupt System

The process of setting up and using interrupts on the PIC18F27Q43 looks very much like any other processor, but the details are admittedly always very confusing no matter what processor you’re working with. You start by following the same procedure as you did for working with the Timer and the DAC, namely reading the appropriate section in the user guide, but in this case what you need to decipher is well beyond the scope of this activity. Very often it is easiest to find a working example and copy that into your project. However, you need to make sure it is working properly, so there’s some effort in checking what you copy, comparing it to what the datasheet tells you, and then verifying it with the debugger.

Since this is fairly complicated, a project has been set up for you. Create a PIC_Activity7 branch from your “Master” branch and then download and copy in the whole project that is provided with this assignment. All of the files are ready to proceed from this point. At the end of this document are some details on how the interrupt system was verified but it’s an optional read.

The parts we’ll focus on are:

1. Interrupt service routines – the functions that automatically run when an enabled interrupt happens.
2. Enabling the interrupt systems and individual interrupts.
3. Managing the interrupt flags.

We will write interrupts for Timer2 (for a much better 1ms system tick) and for Timer1 (for a much better audio generator).

Open interrupts.c and find InterruptSetup().

```
void InterruptSetup(void)
{
    /* Interrupt configuration (MVECEN must be SET/ON in CONFIG3) */
    INTCON0bits.IPEN = 1; //

    /* Enable interrupts */
    INTCON0bits.GIEH = 1; // Enable high priority interrupts
    INTCON0bits.GIEL = 1; // Enable low priority interrupts
```

```
} /* end InterruptSetup() */
```

These 3 lines of code setup the interrupt hardware to use the priority system and enables both high and low priority interrupts. The “GIE” bits are “Global Interrupt Enable” bits which turn on or off all of the associated interrupts that are enabled. If the GIE bits are not on, then no individual interrupts will trigger the processor.

Timer2 Interrupts

Look at the Timer2 interrupt handler. Other than the syntax of the function declaration, you should be able to see exactly what this does. The key line of code is clearing TMR2IF which gets set every time Timer2 counts 1ms. The setting of this bit is what triggers the processor hardware to look at the vector table and jump to the TMR2_ISR function. Interrupt service routines are never called with an explicit line of code like other functions – it’s always the processor hardware that initiates the function call. If you don’t clear the flag, the code will immediately go back to the TMR2_ISR when the function exists and will be stuck forever like that.

```
void __interrupt(irq(IRQ_TMR2), high_priority) TMR2_ISR(void)
{
    /* Clear the interrupt and sleep flags */
    PIR3bits.TMR2IF = 0;
    G_u8SystemFlags &= ~_SYSTEM_SLEEPING;

    /* Increment the system tick timer variable */
    G_u32SystemTime1ms++;
} /* end TMR2_ISR */
```

Timer2 will be configured for a 1ms period, so the TMR2_ISR will run exactly every 1ms regardless of what is happening in the processor. This will be the signal to kick out of the “sleep” function in main, so you can see a global flag “SYSTEM_SLEEPING” is managed here. The global 1ms counter is also incremented. To keep this function fast, the 1s timer is not used. How many days will a 32bit millisecond counter be good for before it wraps around? If you want to see Timer2 configuration, look in void SysTickSetup(void). It’s basically the same as what you did for Timer0, though there’s a “count up to” feature with Timer2 that’s handy. Note that the last thing in is to set the interrupt enable bit, TMR2IE.

```
void SysTickSetup(void)
{
    G_u32SystemTime1ms = 0;
    G_u32SystemTime1s = 0;

    T2PR = 125;           // Match register for 1ms period
    T2CLKCON = 0x01;       // b'00000001' Fosc/4 input
    PIR3bits.TMR2IF = 0;   // Make sure interrupt flag is clear to start
    T2CON = 0xF0;          // b'11110000' Timer on, 1:128 prescale, 1:1 post
    PIE3bits.TMR2IE = 1;   // Enable Timer2 interrupt: SysTick is now running
} /* end SysTickSetup() */
```

$$(2^{(31)})/(60*60*24) = 24.8 \text{ days}$$

Timer1 Interrupts

Timer1 will be used to provide the time base between the successive increments to read the sin function table and update DAC1DATL. Timer1 is nearly identical to Timer0 and needs to preload the counter registers to get the correct amount of time. Because there is only a 1:8 prescaler, this timer is going to count 0.5us per tick but that's fine given the frequencies we need to generate.

Since the Timer1 functionality is in user_app.c, and the interrupt handler is in interrupts.c, global variables are used to allow data to be used by both functions. We can talk a lot about global variables and why you shouldn't use them, but often with embedded systems you don't have a choice, especially when timing is critical. Three globals are defined, and the sin table is moved to a full global as well since the ISR needs it.

```
volatile u8 G_u8UserAppFlags;           /*!< @brief Global state flags */
volatile u8 G_u8UserAppTimePeriodHi;    /*!< @brief Global saved Timer1h */
volatile u8 G_u8UserAppTimePeriodLo;    /*!< @brief Global saved Timer1l */
```

Look closely at the definition for InterruptTimerXus. It is very close to TimeXus() that you wrote, with the following updates:

1. A "Continuous" parameter is added so the function can run once or keep running.
2. The maximum time is half that of the TimeXus() due to the 1:8 maximum prescaler. It was decided to simply clip the value to 32767 even if the user tries to pass in something higher. In a real application, some sort of error or message should be communicated.
3. Once the period calculations are made, they are saved to global variables.
4. The Timer1 interrupt is enabled when everything is ready.

```
void InterruptTimerXus(u16 u16TimeXus_, bool bContinuous_)
{
    u16 u16Temp;

    /* Disable the timer during config */
    T1CONbits.ON = 0;

    /* Correct the input parameter if it's too high */
    if(u16TimeXus_ > 32767)
    {
        u16TimeXus_ = 32767;
    }

    /* Double the time so it's in us not 0.5us*/
    u16Temp = u16TimeXus_ << 1;

    /* Calculate, save, and preload TMR1H and TMR1L based on u16TimeXus_ */
    u16Temp = 65535 - u16TimeXus_;
    G_u8UserAppTimePeriodHi = (u8)( (u16Temp >> 8) & 0x00FF);
    G_u8UserAppTimePeriodLo = (u8)( u16Temp & 0x00FF);
    TMR1H = G_u8UserAppTimePeriodHi;
    TMR1L = G_u8UserAppTimePeriodLo;

    /* Flag continuous mode if required */
```

```

G_u8UserAppFlags &= ~_U8_CONTINUOUS;
if(bContinuous_)
{
    G_u8UserAppFlags |= _U8_CONTINUOUS;
}

/* Clear the interrupt flag, enable interrupt and enable Timer */
PIR3bits.TMR1IF = 0;
PIE3bits.TMR1IE = 1;
T1CONbits.ON = 1;

} /* end InterruptTimerXus() */

```

Once this function is called, the system will run Timer1 interrupts. The purpose of Timer1 is to choose the next sin value from the table. Since this is a very application-specific example, the code for this is placed directly in the TMR1_ISR. We saw in the previous activity that taking every 4th value in the sin table generates a nice enough sin wave, so the index can increment by 4. When we get to defining musical notes, we'll take this in to account to set the speed at which Timer1 will interrupt. For now, all you need to do is add the two lines of code in the Event Handler section of this ISR to grab the next sin value for DAC1DATL and increment the index by 4 (you did this in a previous activity).

```

void __interrupt(irq(IRQ_TMR1), high_priority) TMR1_ISR(void)
{
    static u8 u8Index = 0;

    /* Reload the timer - do this first to minimize latency */
    TMR1H = G_u8UserAppTimePeriodHi;
    TMR1L = G_u8UserAppTimePeriodLo;

    /*****
    Handle the timing event here (usually a call-back function)
    In this case, we load the next value of the sinusoid.
    KEEP THIS SHORT!
    *****/

    WRITE YOUR CODE HERE

    /*****
    End of event handling
    *****/

    /* Clear the interrupt flag */
    PIR3bits.TMR1IF = 0;

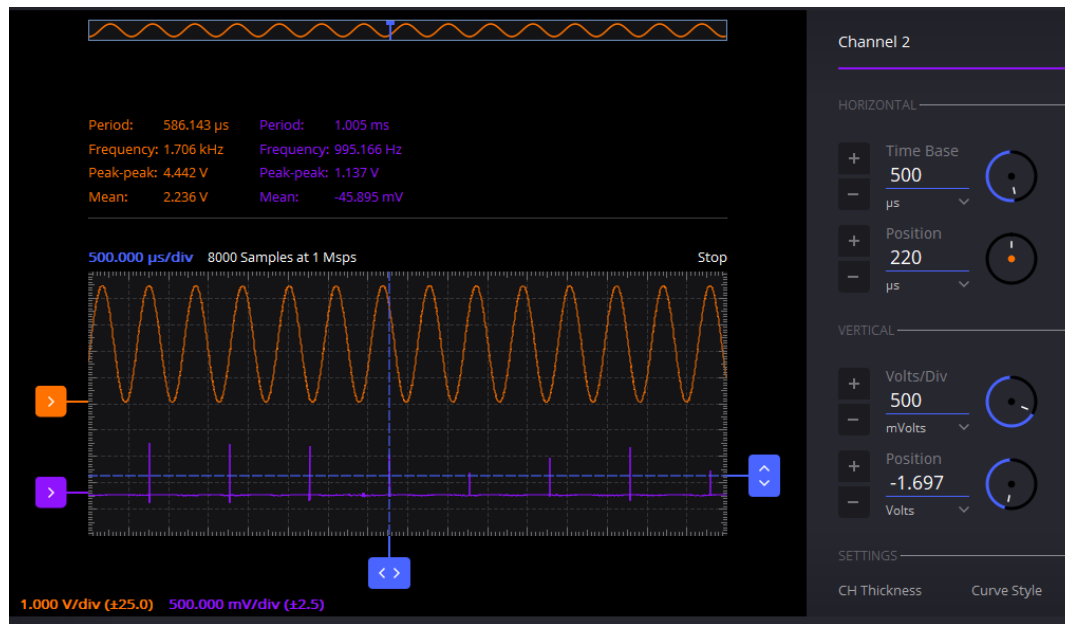
    /* Turn off the timer and interrupt if this is one-shot */
    if( !(G_u8UserAppFlags & _U8_CONTINUOUS) )
    {
        PIE3bits.TMR1IE = 0;
        T1CONbits.ON = 0;
    }

} /* end TMR1_ISR */

```

You can see the ISR handles the “continuous” case. If the function is only to run once, the interrupt is simply disabled inside the ISR.

At this point, you have written two lines of code and that should be enough to test the audio. Add a call to `InterruptTimerXus()` at the end of `UserAppInitialize()` to set a 1kHz time that runs continuously. Use `Scopy` to look at both the system tick and the 1kHz sinusoid. Take a screen shot of this and make sure you can hear the tone on your speaker. The sinusoid in the screen shot below is too fast for unknown reasons at this point... yours should be, too, as this is from the code download.



Programming a Song

With the code that generates sound in place, your mission is to program your PIC to play a song. Notice a file call `music.h` which has a bunch of constants for music. The note values are specifically set for the considerations of the Timer1 design. It's suggested to keep it simple so you don't have to define a bunch of new notes in `music.h`.

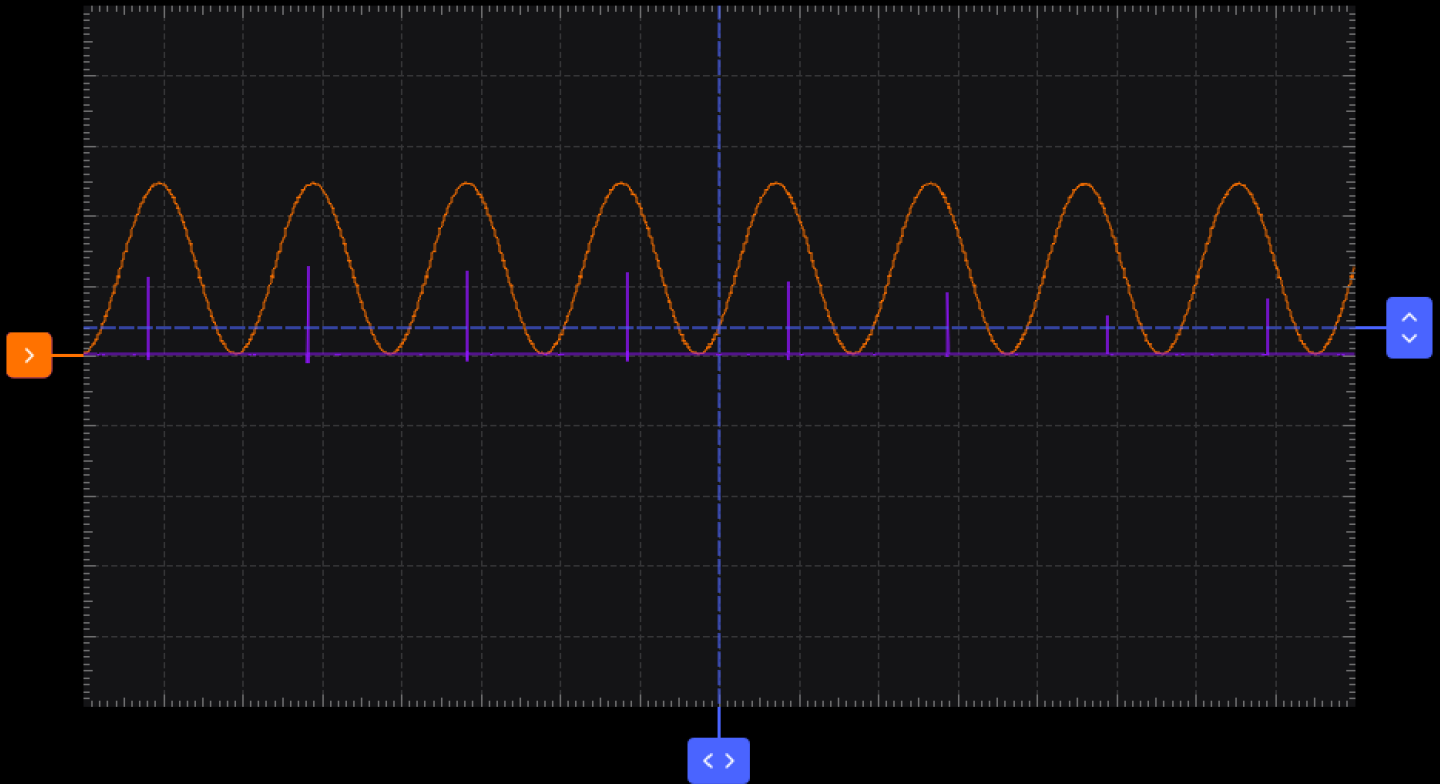
Below is the sheet music for Twinkle, Twinkle Little Star which was the piece chosen to demo in this exercise. If you don't know how to read sheet music, Google it. The letters above the notes are chords which you can ignore. The notes in the first line of the song per music.h are (C4, C4, G4, G4, A4, A4, G4, F4, F4, E4, E4, D4, D4, C4). The solid notes are "quarter notes" (N4) and the hollow notes are "half notes" (N2) which determines how long they should be played.



Period:	971.125 μ s	Period:	272.000 μ s
Frequency:	1.030 kHz	Frequency:	3.676 kHz
Peak-peak:	4.889 V	Peak-peak:	2.796 V
Mean:	2.439 V	Mean:	49.233 mV

500.000 μ s/div 8000 Samples at 1 Mps

Stop



2.000 V/div (± 25.0) 2.000 V/div (± 25.0)

Twinkle Twinkle



Source: <https://www.music-for-music-teachers.com/twinkle-twinkle.html>

Since all of the timing and interrupts are done, this becomes just another programming exercise. You are essentially writing the algorithm for MIDI sequencing. You can look up MIDI if you wish, but generally a MIDI file consists of notes and durations of those notes. A computer reads the note to play and plays it for the duration specified.

Implement the song of your choice (use Twinkle, Twinkle if you want). The first line is good enough. The notes should be an array, and the note duration should be in another array. All code should be in `UserAppRun`. You do not have to do anything with the interrupts or timers. All you need to remember is that `UserAppRun` is entered every 1ms, so your code just has to use this to time out the note duration and then make the note change. There's a special case of no note. You will also notice that when you first do this, you will not be able to distinguish between two of the same notes played in succession unless you turn off the note near the end for a short period of time. The constant `"REGULAR_NOTE_ADJUSTMENT"` in `music.h` provides a nice amount of time for this.

Make sure your song has a short delay at the end before repeating. Record a video of your song playing and include it in your git commit..

What to Hand In / Review

Save your design documentation and screen shots in a single document in your source code folder and commit it and your code. Don't forget to push it to Github. Perform a code review on someone's code from your learning community who you have not already done a code review for. Upload the code review spreadsheet to the D2L Dropbox for the assignment and be sure to email the person whom you reviewed a copy.

Read this if you REALLY want to know how to set up interrupts on the PIC

It took many hours to figure out and verify how to setup interrupts on the PIC18F27Q43. There is an example of interrupt configuration, but it's the old-style of PIC interrupt system that does not use the vector table and multiple priority levels.

Instead the example code from the datasheet was copied in and tweaked until it worked. Some of the symbols in the datasheet were not correct for the current version of the XC8 compiler, so those had to be determined and corrected. To make sure that things were working, the code was build at the program memory was examined to see if things were where they should be.

Note it's often WAY easier to do this in assembler, because each compiler has highly-specific syntax to make it generate the correct assembler code for an interrupt. These are really just macros that end up writing a bunch of assembly language based on the information provided. It turns out that to create an interrupt handler and put its address in the vector table, the "function" definition for the ISR looks like this:

```
void __interrupt(irq(default), low_priority) DEFAULT_ISR(void)
{
}
```

This is a macro, not really a line of C code though it looks and behaves like a function call. The "__interrupt" tells the compiler it's an ISR, which in turn will write the appropriate special assembly language including the "retfie" (return from interrupt) instruction when the function ends. It also adds the basic context saving code to save critical registers that must be preserved and restored when the interrupt is done so that the main program continues to run where it left off. Once we had this working, the following was checked:

1. Find MVECEN – see that it's a configuration bit and confirm that it is set in configuration.h (line 42 where the CONFIG3 register definition is made).
2. Confirm IVTBASE is 0x0008
3. Confirm that the ISR addresses are appearing in the vector table and find out where the ISRs are actually being put in flash to confirm they are loaded properly.

To find the ISR addresses, we put a breakpoint on TMR2_ISR and looked at the disassembly window to check its address: 0x300

The datasheet indicates that the TMR2 interrupt vector number is 0x1B. Section 11.3.3 in the datasheet gives the formula for calculating the address of an ISR.

11.3.3 Interrupt Vector Table Address Calculation

MVECEN = 0

When the MVECEN Configuration bit is cleared, the address pointed to by **IVTBASE** is used as the high-priority interrupt vector address. The low-priority interrupt vector address is offset eight instruction words from the address in IVTBASE.

For PIC18 devices, IVTBASE defaults to 000008h, hence the high-priority interrupt vector address will be 000008h and the low-priority interrupt vector address will be 000018h.

MVECEN = 1






Each interrupt has a unique vector number associated with it as defined in the IVT. This vector number is used for calculating the location of the interrupt vector for a particular interrupt source.

Interrupt Vector Address = IVTBASE + (2*Vector Number).

So we take the vector table base address (0x0008) and add twice the vector number:

$$0x0008 + (2 * 0x001B) = 0x3E$$

We look in the Program Memory window at address 0x0003E and see it holds the value 0xC0

Output	Usages	Notifications	Breakpoints	Variables	Call Stack	Search Results	Program Memory x
	Line	Address	Opcode	Label	DisAssy		
	31	0003A	00AF				
	32	0003C	00AF				
	33	0003E	00C0				
	34	00040	00AF				
	35	00042	00AF				
	36	00044	00AF				

Section 11.3.2 of the datasheet tells us that the location of the ISR is the entry in the vector table x 4.

11.3.2 Interrupt Vector Table Contents

MVECEN = 0

When MVECEN = 0, the address location pointed to by **IVTBASE** has a GOTO instruction for a high-priority interrupt. Similarly, the corresponding low-priority vector also has a GOTO instruction, which is executed in case of a low-priority interrupt.






MVECEN = 1

When MVECEN = 1, the value in the vector table of each interrupt points to the address location of the first instruction of the Interrupt Service Routine, hence: ISR Location = Interrupt Vector Table entry << 2.

So we check:

$$0xC0 \ll 2 = 0x300$$

And see that indeed that is 0x300 where our TMR2_ISR breakpoint stopped.

Output	Usages	Notifications	Breakpoints	Variables	Call Stack	Search Results	Program Memory
							
							
							
							
	Line	Address	Opcode	Label	DisAssy		
	385	002FE	FFFF	__end_of_SystemSleep	NOP		
	386	00300	96B1	_TMR2_ISR	BCF 0xFB1, 3, ACCESS		
	387	00302	9C0F		BCF 0xF, 6, ACCESS		
	388	00304	0E01		MOVLW 0x1		
	389	00306	2609		ADDWF 0x9, F, ACCESS		
	390	00308	0E00		MOVLW 0x0		

From there we had the confidence that the interrupt setup was correct and we could focus on the rest of the program.