F28069 MICROPROCESSOR TUTORIAL

PWM Interrupts and Sine Wave Generation

by

Ali Hadi Al-Hakim

March 2013

Revision 2

For feedback on this tutorial please contact aa6909@ic.ac.uk

The following tutorial was made using Code Composer Studio Version 5.

1 USING PWM INTERRUPTS

1.1 The PIE Vector Table

To use interrupts we must first enable the data path of the microprocessor that will allow these to occur. The figure of page 160 in the Technical Reference Manual shows us some of this.

There are five bits which must be set to allow an interrupt to get from the peripheral (e.g PWM, ADC) into the CPU where it will then be executed. These are the PIEIFR flag, PIEIFR enable, PIEACK enable, IFR flag, IER enable.

In addition, we must also define each of the interrupts in the PIE vector table. The PIE vector table is basically the system used on this microprocessor to prioritise and use interrupts. Fortunately, most of this code is provided to us by TI in the original files. The functions that will enable all of this are provided to you in the Pie_Control_Init.c and Pie_Vector_Table.c C-code file. Have a look inside these files and add the functions in the, (three in total) to the initialisation part of your main() function. The order in which you add these does matter. For example, PwmInit() already in code your should be called after you call these new functions.

When you add the functions to your main(), and then try and run the code you will get some warnings stating that a function has been declared implicitly. This basically just means you've called the functions without clearly defining them beforehand. To overcome this you need to write a function prototype, so when compilation occurs, the system knows this function exists somewhere in the code. This is the reason the header file Function_Prototypes.h exists. Open this file and add the function prototypes in the format:

```
extern void [FunctionName](void} ;
```

Lastly, there is another assembly function that you should add to allow the interrupts to work. I'm not sure exactly why this is required or what it does but it is necessary. I think that it's used for interrupt nesting, which I think means stacking up interrupts after each other. (wiki link)

```
asm(" CLRC INTM, DBGM");  // Enable global interrupts and real-time debug
```

Add the above line to the last of the main() initialisation section.
With these new initialisation functions executed, you should now be able to begin using interrupts for certain peripherals.

However, lastly you'll also find you have an error for unresolved symbols. This is to do with the fact that in Pie_Vector_Table.c there is a function that is called PieVectTableInit and uses symbols that are not present in your code. The reason for this is simply you need to add another TI file. The file provided is the one that contains all of the Interrupt Service Routine (ISR) functions. You have also been given a C-code file called Interrupt_SR.c. Add this to your project and debug again. It should run now with no warnings or errors.

If you go into Interrupt_SR.c you'll notice it's just a list of functions. The unique thing about the ISRs is simply how they will be called. Normally you will type the line

```
  [FunctionName] ( [Arguments] );
```

to call a function as you have done in main(). However, the ISRs will be called by something else in the microcontroller. For example, you can set them to be called when a PWM signal reaches a certain point in its count (ZERO, PRD, CMPA/B), or based off a timer so that every t seconds the ISR will call, or off the sampling of an analogue signal at one of the ADC peripherals.

1.2 PWM Interrupts

The first interrupt you will call will be based off one of the PWM signals that you generated. You have already written some of the code to enable this. We are going to use this interrupt to generate a Sine wave signal.

Looking at the Technical Reference Manual you'll find that the interrupt for PWM1 is in the PIE group 3. Open Interrupt_SR.c and Ctrl-F "3.1". This should take you to where we want to be; an ISR called EPWM1_INT_ISR(void)

You'll notice there is some code/comments already in the interrupt function.

```
  // Insert ISR Code here

  // To receive more interrupts from this PIE group, acknowledge this interrupt
  // PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;

  // Next two lines for debug only to halt the processor here
  // Remove after inserting ISR Code
 __asm ("      ESTOP0");
  for(;;);
```

The code currently in the ISR is not much use. It's implemented to simply allow you to test if you'll enter the interrupt when you run the code. `_asm ("   ESTOP0")` is an assembly language function which will cause the system to stop when it reaches this line of code (essentially a software breakpoint). `for(;;)` is another way of creating an infinite loop, much like `while(1)` does.

Debug the code and run it. You should find nothing new is happening yet. The code is simply running through the while() polling loop and generating your PWMs but no interrupts are happening. You can prove this by pausing the code. What you'll find when you pause it is you always stop in the while() polling loop.

This is happening because you haven't yet enabled the PWM1 interrupt. Stop the code and let's change this.

When given the code initially you were provided with the lines:

```
// --- Event triggers
EPwm1Regs.ETSEL.bit.INTSEL= ET_CTR_ZERO;      // Event occurs when TBCTR == ZERO
EPwm1Regs.ETPS.bit.INTPRD = ET_1ST;           // Generate an interrupt on every event
EPwm1Regs.ETSEL.bit.INTEN = 0;                // ePWM2 interrupt disabled
// --- Enable the EPWM1 interrupt (3.1)
PieCtrlRegs.PIEIER3.bit.INTx1    = 0;         // Disable EPWM1_INT in PIE group 3
```

These are the lines of code which will initialise a PWM event to cause an interrupt. At the moment this code tells us: an event will occur every time the counter reaches 0, but the interrupt event is disabled. There are two lines which are disabling the interrupt. The first is `ETSEL.bit.INTEN` which is outright inhibiting the generation of an interrupt signal from the peripheral when an event occurs. (where an "event" is every time the PWM counter becomes equal to zero as defined by the top line). This line of code maps to the box labelled "From Peripherals of External Interrupts" on page 160 of the Technical Reference Manual. The second line of code needed to enable the interrupt is the last one in the block above. This controls whether this particular interrupt signal is allowed to propagate through the PIE controller. I believe this line maps to the switch PIEIERx(8:1) of page 160 of the Technical Reference Manual. You can see in the diagram, if disabled the interrupt signal will not propagate to the CPU, whereas if enabled, the switch is closed and the signal gets one step closer to the CPU, where it will be executed.

Figure 1-84. Overview: Multiplexing of Interrupts Using the PIE Block

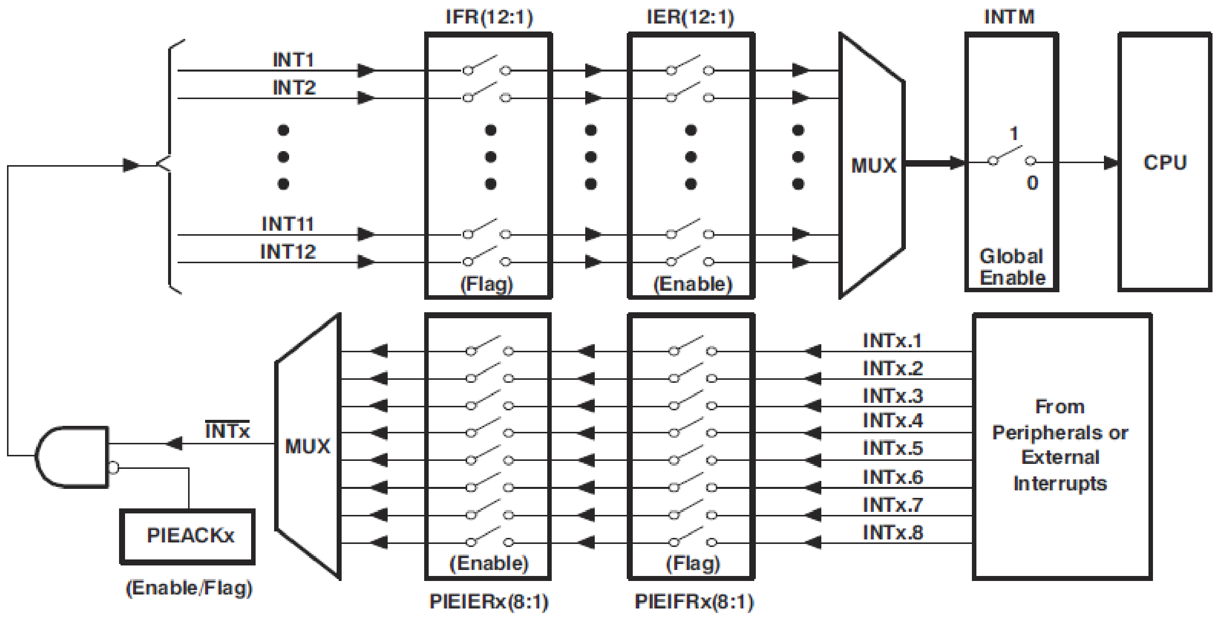


Diagram is taken from page 160 of the Technical Reference Manual and shows the interrupt propagation path from peripheral, through the PIE controller and to the CPU.

We want to enable the EPWM1 interrupt so change the two bottom lines of code to equal 1 to enable them.

If you are still not entering the ISR, make sure that you check your PWM signal IS enabled in the DeviceInit.c file.

Once you have it working you'll want to modify the code in the ISR. First, change what's inside the ISR to only :

```
__interrupt void EPWM1_INT_ISR(void)          // EPWM-1
{
      PieCtrlRegs.PIEACK.bit.ACK3 = 1;        // Acknowledge the interrupt
      __asm ("      ESTOP0");
      EPwm1Regs.ETCLR.bit.INT    = 1;         // Clear the interrupt flag
}
```

The two lines around the software breakpoint simply enable this interrupt to be accessed again. The top line acknowledges that the PIE group has been accessed, clears the flag (re-enabling the group) and sends the interrupt signal through toward the CPU. The bottom line of code will clear the PWM1 interrupt flag allowing it to be flagged again and again. If you don't add this in, you'll only ever enter the ISR once.

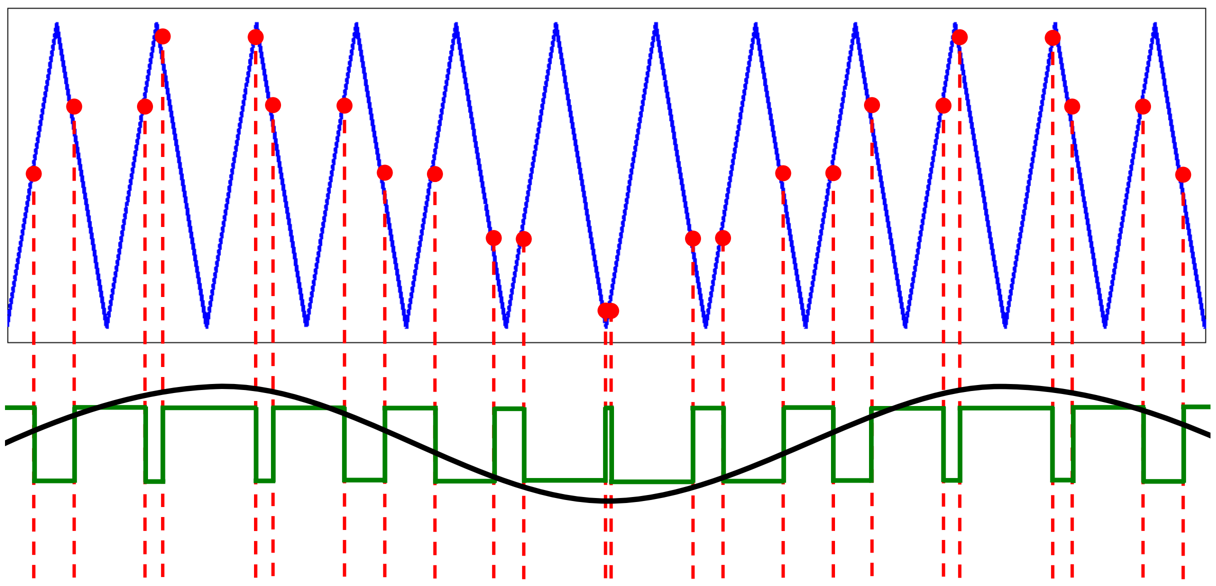
In our case we want to re-enable the flag just before we leave the ISR so that we know it will be ready next time. However, in some cases, it may be beneficial to not reset this interrupt until perhaps another event has occurred after which you would reset it and enable the interrupt.

Re-run the code and you should now find that when you press play you'll automatically get stopped in the ISR by the software breakpoint; and when you press play again, the same will happen.

To time the rate at which you enter the ISR, go to "***Run*** -> ***Clock*** -> ***Enable***". After doing this you should notice a small clock icon appear at the bottom of the CCS window. Run your code with the ISR active and you'll notice the clock counts. The number it gives is the number of cycles that have passed between when you restarted the code and when it stops. To reset the clock just double click on the icon. For your 20kHz PWM1 interrupt you should find that there are roughly 4,000 cycles between each call of the ISR. Noting that you've set your system clock to 80MHz, this tells us we are accessing the ISR every:

$$\frac{4{,}000}{80{,}000{,}000} = 50\mu s$$

As we have set our PWM interrupt event to happen on every zero count using a PWM of frequency 20kHz, this is exactly as we expect. Note that $50\mu s$ is the time period for a 20kHz signal.



For more information on the steps to set up an interrupt, check out page 168 of the Technical Reference Manual.

2 GENERATING A SINE WAVE

We're going to use the interrupt you enabled in the last section generate a 50Hz sine wave PWM signal. The way in which we do this is to continuously change the duty-cycle of the PWM signal which will give a varying-duty-cycle pulse train. It should look something similar to that shown in the diagram below if we could capture the changing signal. By filtering this signal, a sine wave can be achieved. You can even see conceptually the sine wave in the pulse train as shown in the figure below.

When the duty-cycle is full, you have the sine wave peak and when it is zero, the sine wave trough. The red dots in the image represent the compare values (CMPA/B) we use to modify the PWM signal's duty-cycle as discussed in the tutorial on PWM waveforms. The main issue is we must choose the compare values very carefully to actually get a sine wave and for a good sine wave it is not simply a case of arbitrarily choosing different places to set or clear the PWM output.

The correct compare positions must be generated using a sine-look-up table. This is literally a table of sine wave values. The code must look-up the necessary value for a particular point in time and then change the compare point. This should happen once every cycle.

Conceptual diagram of generating a sine wave from a two level PWM pulse.

We already know now that we could update the compare once every cycle by using the PWM1 ISR. Thus, the only step we now need is to generate the sine look-up table.

There are several steps to this but all is provided by TI and now provided to you.

In the tutorial folder you should find another folder called "dsp" . Go through its contents until you reach a list of more folders. Inside "doc" you will find instructions on the available sine wave generation functions, inside "lib" is another library you should add to your linker and finally you should add the "include" folder to your included files.

You should add the library "C28x_SGEN_Lib_fpu32.lib" (See "Tut1- Project Creation and PWMs" on where to add these if you don't remember).

Now, with the list of new C-code files provided there should be one called **SineRefInit.c**. Add this to your project and open it. Go to page 53 (PDF page 58). We are interested in the function which will create two sine wave outputs for two 180 out of phase waveforms.

2.1 The Look-up Table Generation Function

You are going to use the function detailed in page 53 of the sgen_mdl.pdf document which can be found in the "..\dsp\SGNE\v101\doc". Open **Sine_Ref_Init.c** and you'll see the structure described on page 53 is supposed to be initialised here.

To generate your sine wave you must first describe exactly what you want your sine wave to be like and this is where you do it. The internal structure variables are fairly self explanatory by their names. Go ahead and initialise the sine wave you want (use the manual or the comments given to help)

Offset = $(0)_{\text{dec}}$
Gain = $(0)_{\text{dec}}$
Frequency = $(50\text{Hz})_{\text{dec}}$
Phase = $(180^{o})_{\text{dec}}$

Note: You can give the values in decimal or hexadecimal format but be wary that these variables may be signed (i.e. $0 \neq 0x0000$*) (check page* 58 *to know!)*

`sinGen.step_max` is perhaps the only non-obvious parameter. This is basically used to determine how accurate our sine wave will be to the desired frequency as it will set the frequency resolution of the sine look-up table via the equation:

$$freqResolution = \frac{maxFreq}{step\_max}$$

Where maxFreq is something we define as the maximum possible frequency you would want to generate (i.e. the maximum value you would use as your frequency reference). Well, you are only actually interested in generating a 50Hz sine wave, or perhaps 60Hz, so let's just go a bit above that and say maxFreq = 65Hz.

You should end up with something equivalent to:

```
void SineRefInit(void)
{
	sinGen.offset		= 0x7FFF;
	sinGen.gain		= 0x7FFF;
	sinGen.step_max		= 0xD4FDF4;
	sinGen.freq		= 0x62762762;
	sinGen.phase		= 0x80000000;
}
```

Actually, this code is not that great to work with. It isn't flexible and therefore, if you were make changes in the code later you may have to remember to make your way all the way back to this C-code file to also modify it. More so, if someone else ever modified your code they may not even know they need to come and make changes here too to make everything work properly!

You don't need to change it now but you should think about how you would change the code here so that it's 'aware' of other code changes and will respond accordingly.

There are next just a few more steps to make to get the sine look-up table working. You have to:

- Define and initialise your variables and structures
- Declare memory space for the sine look-up table

2.1.1 Define and Initialise

I have typically defined and initialised my variables/structures in two different files. Defining something is to just explicitly state in the code that this variable exists. Then to initialise it is to assign a value to the variable before it is used in the programme. The files I've done this in for this case are:

`Sine_Ref_Decl.h`	contains the definitions of variables used in Sine_Ref_Init.c.
`Global_Variable_Init.h`	contain initialisations for every variable used in your programme.

You will need to include (use `#include`) `Sine_Ref_Decl.h` in Function_Prototypes.h with the other headers that are included here. On the other hand, include `Global_Variable_Init.h` in main.c. The reason you put `Global_Variable_Init.h` here is that this stops it from being called multiple times and therefore repeat initialising functions by accident which normally sends "redefined" errors at you.

These files are provided to you and you should just be able to uncomment the lines you'll need to make your current build work. If you have trouble executing the code then check that you've definitely included all of the new linker, include and library search paths and files. Following this, check the order in which you're calling functions or including header files in the Function_Prototypes.h file.

2.1.2 Declaring Memory Space

Now when you build or debug you should hopefully come up with a no-error situation. However, you should have a single warning that says something about "creating output section "SINTBL"". This is to do with the fact you're trying to build a sine look-up table yet you haven't yet informed the system where it can save all of this data as it's treated differently than say a variable that you may otherwise create yourself. To do this you must explore one of the .cmd files.

Open `28069_RAM_lnk.cmd` and have a look through it. What you have here is where aspects of the system are being allocated certain registers (or memory). In the MEMORY area each block has a beginning memory address and then a length defined in hexadecimal. After this is a SECTIONS area. It is here that we will be adding a line of code to place our SINTBL into a space in memory.

To be honest, I don't remember a lot about this other than you write in a line of code similar to those already in it, but for SINTBL. I had to do a little research about where I can put it but don't remember much about it other than you DO NOT put it in the "memory" part. You must put it in the part labelled "SECTIONS" (line 106).

Scroll down to line 126 of this file and make some space. You're going to allocate your memory here into one of the RAM blocks of memory. Add the line:

```
   SINTBL                    : > RAML7,    PAGE  = 1
```

This is now basically saying to place the sine look-up table into the memory block RAML7 which is in page 1 of the memory section. Something like that.

Now hopefully, when you run your code, there should be no problems and you're ready to generate your sine-wave PWM waveform.