

F28069 MICROPROCESSOR TUTORIAL

Project Creation and PWMs

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.

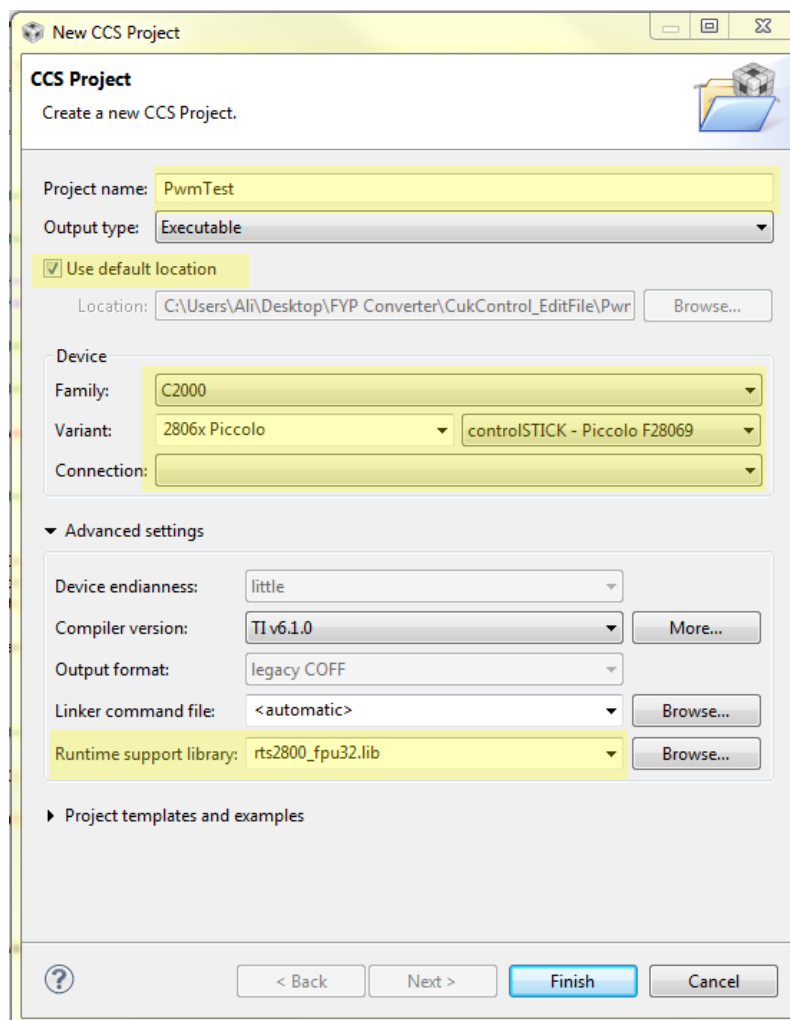
1 USING CODE COMPOSER STUDIO (CCS)

1.1 Creating a Workspace

Install the latest version of CCS from http://processors.wiki.ti.com/index.php/Download_CCS.

Once installed and when you run CCS you will be asked to open your *workspace*. Create a folder called F28069_Test or something similar. Now open access this folder as your workspace. A workspace contains your *project* folders, which contains your code.

When CCS opens into your defined workspace you will need to start a new project. Go to **File -> New -> CCS Project**. Insert the following settings:



Click Finish to continue.

This will now open your main.c file. However, there are a few more things you will need to do before you can start coding.

1.2 Target Configuration

The next step is to set up a target configuration. This essentially is used to inform the IDE what device it is connecting to and how to interface with it. If you don't set this up, when you try to compile the code for debugging you will just get an error message. To set up a target configuration, go to **File -> New -> Target Configuration File** which will open a new window.

Type in a file name for the target configuration and make sure it ends with the extension **.ccxml** and leave the "Use shared location" checkbox BLANK. Click Finish

You will be greeted with a new screen. Enter the following settings:

Basic

General Setup
This section describes the general configuration about the target.

Connection: Texas Instruments XDS100v1 USB Emulator

Board or Device: control

- ☐ Developer's Kit - Dual Motor Control and PFC (F28035)
- ☐ Developer's Kit - Motor Control and PFC (F28035)
- ☐ controlSTICK - Piccolo F28027
- ☒ controlSTICK - Piccolo F28069

Piccolo F28069 controlSTICK


Note: Support for more devices may be available from the update manager.

Click save and close the target configuration tab if it stays open.

1.3 Linker and Include Files

The next step is to connect all of the necessary include and linker files to your project. These contain libraries, variables, register name etc. required to let the code work with your microcontroller from the get go. If you don't connect these then you won't be able to properly connect to your device.

To include the necessary files, click on your project folder in the left-hand-side navigation bar, to activate it, and then go to **File -> Properties**. In the drop down menus on the side, find the "Include Options" page under C2000 Compiler. You will also need to add some files to the "File Search Path" page under C2000 Compiler. (*This may be different for different versions of CCS*).

You have been given a folder called "support" which contains all the files that you should need for now. When on the "Include Options" page click on the green plus sign () of the bottom window and add the following search paths:

```
"YOUR ROOT\support\include\ccsv5-include"  
"YOUR ROOT\support\include\devkit-include"  
"YOUR ROOT\support\include\devsupport-com-include"  
"YOUR ROOT\support\include\devsupport-hdr-include"
```

Next, find the "File Search Path" page and click on the green plus sign of the bottom window to add the following:

```
"YOUR ROOT\support\library\ccsv5-lib"
```

Lastly, on this same page, click the green plus sign of the top window and add:

```
"libc.a"
```

An important point to note here is when you include these files or search paths, do not forget to include the speech marks around the inputs.

1.4 Command Files

The last thing you need to do before you can start coding is include another .cmd file which is used to define some of the memory mapping for the device. For this basic example you'll be saving everything to the RAM. However, if you look at the MHPaC (Micro Hydro Plant and Controller) code I provided you'll notice this is actually running in flash. The flash is slower but has more memory available. For now though, add the second .cmd file to your project folder. You can find this in the "support" folder provided to you.

To do this, find the file **F2806x_Headers_nonBIOS.cmd** in "..\support\cmd files" and copy it into your project folder. You can do this by dragging the file into your project folder in or out of CCS. Just ignore the other.cmd files also in the folder.

2 INITIAL STEPS

2.1 Adding files


You have been provided with four .c C-code files. These are:

main.c	A fairly empty file with some basic c code inside
Device_Init.c	Contains the DeviceInit() function to initialise some settings
Gpio_Init.c	Contains the GpioInit() function to initialise pins
Pwm_Init.c	Contains the PwmInit() function to set up PWM outputs


You have also been given one .h header files, which are:

Function_Prototypes.h	Contains all project function prototypes
-----------------------	--

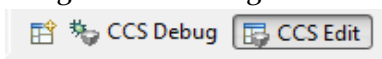
Please note, that the way in which I've made these folders may not necessarily be the correct way. I'm by no means an experienced programmer and this has all been written in a way that suited me. Nonetheless, it should be readable and relatively easy to work with.

Now, if you click the hammer icon () you can build the project even without a microprocessor connected to see if there are any errors. You will find that now there are some. The error that occurs refers to some undefined symbols. If you look in the "Console" window you can see which symbols have not been defined.

The reason for this error is there is still a file that needs to be added to the project which includes the definitions of these symbols. This is again a file that is actually provided by TI deep somewhere in the files provided by TI. For your convenience you will find the file you want in "..\support\source files\devsupport-hdr-source". Add this .c file to your project.

Now when you build the project you should find there are no errors. If you have a microprocessor plugged in then you can click the green bug icon () and actually compile it onto the microprocessor.


If you click the green bug icon you will enter the debug mode in CCS. This will just change around the interface a bit but it mostly will all behave the same. You can still edit and re-debug code in this mode, the only limitation is you don't have easy access to all the files in your project as the left-hand navigation bar has gone. To switch between the modes use the tabs in the top-right corner of

CCS ().

If you want you can explore some of these files to see what treasures they hold. However, we'll mostly be going through most of them as this tutorial progresses anyway so don't spend too much time doing it.

2.2 Getting a response from the microprocessor

The following tutorial will be written as if you have an F28069 ControlSTICK to debug and therefore watch respond to your code command. If you don't yet have one can still follow the steps, but of course you won't be able to run the tests. Nonetheless, you can still write the code and test it afterwards.

When in debug mode you can press the green arrow () to run the code. By default you will begin running from the very beginning of the main() function in main.c.

If you run the code you will notice nothing happens. Let's get a response from the microcontroller so that we know it's working.

You can already see in the main function there is some code that should be working. It looks like:

```
while(1)
{
    if(CpuTimer0Regs.TCR.bit.TIF == 1)
    {
        CpuTimer0Regs.TCR.bit.TIF = 1;           // Clear flag
        GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1; // Toggle GPIO34 (LED)
    }
}
```

Firstly, there is just an infinite while loop. This just causes the program to continuously run through the code inside the loop. It basically says, "while 1 = 1 stay in the loop". Clearly, 1 will always equal 1, so the loop is infinite.

Inside the infinite while loop (also called the polling loop) we have a small bit of code and your first introduction to registers; something you will soon see a lot of. This small block of code says:

"If the flag for the CPU timer 0 peripheral has been set, then:

1. Clear that flag so that it can be set again in the future, and
2. Toggle the general purpose input/output (GPIO) pin 34. "

You will notice on the microprocessor board an LED component called LD2. This is actually connected to GPIO34. If we are toggling GPIO34 then we would expect to see the LED flashing on and off, yet we don't.

This is an initialisation problem!

As this is related to a GPIO pin, go to Gpio_Init.c. Take a look through this list and see if you can find why there is no output signal coming from GPIO34.

Once you've figured it out and changed the code, re-debug and then run your program. If you found the correct solution you should find the LED is now flashing. This is pretty useless right now but it is nice to see something is working. When your code gets more complex, these few lines can also be useful to check that a certain area of code is actually being accessed without the need for breakpoints.

Now take a look at the line of code above the polling loop.

```
CpuTimer0Regs.PRD.all = mSec100;
```

This is another register related to CPU Timer 0. In this case, it is used to set the period of the timer. It is currently set to 100ms but you can change it. Left-click on "mSec100" so that the cursors rests in it, then right-click and select "Open Declaration". This will open up one of the files we included earlier in the tutorial and will show a set of timer period options for you to choose from.

Try changing the CpuTimer0Regs period setting to something else and run the code. Noting that the default system clock is 80MHz (80,000,000), can you work out how this timing is working?

You now have the LED flashing and have learnt some basics on register based programming. Let's move on to the PWM signals now and create some more interesting signals!

3 PWM EXAMPLE AND EXPLANATION

In the C-code file "Pwm_Init.c" which has been provided to you, you will find a fully working, very basic PWM signal.

This tutorial will only briefly go over what each of these lines do. To fully understand what is happening you are expected to have a look at the TMS320x2806x Piccolo Technical Reference Manual (unfortunately, I can't find a proper web link to this anymore so I've provided a .pdf version I have anyway).

The Technical Reference Manual is your best friend from now on and you should look at it. Go to page 244 for descriptions of the PWM functionality and page 329 for the start of the register settings information.

3.1 Background

First off, notice the line:

```
asm(" EALLOW");    // Enable EALLOW protected register access
```

This you will see quite a lot wherever we are modifying registers. It's an assembly command that allows you to access some of the registers which would otherwise be restricted. At the end of the PwmInit() function you'll see there is also "asm(" EDIS");" which re-restricts access.

Moving on.

The C2000 Piccolo F2806x series of microprocessors provide 8 EPWMs, each with two outputs (A and B) outputs that can be different but will share some similar properties. However, on this development board you will only have access to the outputs 1-4A and 1-4B (see "Pin Layout.png"), the rest are only useful for internal timing actions within the microcontroller.

This first example PWM you are given is set up through peripheral EPWM2. It is a 5kHz pulse train generated from a triangle wave with a constant duty cycle of 50%. Only output A is being used and output B ignored. There is also no dead-time set up between these two signals. The interrupt action for this peripheral has also been disabled although the code is present to easily enable it. Outputs EPWM2A/B can be measured from pins 27 and 23, respectively, as shown in "Pin Layout.png".

3.2 Frequency

On Line 33 you can see that the EPWM1 time-base has been set to 8000.0 (It has been written with a decimal point to initialise it as a *float* which is important for some cases). Look at page 249 of the Technical Reference Manual to understand how the frequency is set. Note that lines 35 and 36 are setting the value of TBCLK.

```
EPwm2Regs.TBPRD          = 8000.0;    // Set timer period
EPwm2Regs.TBCTL.bit.CLKDIV = TB_DIV1;  // No pre-scaling
EPwm2Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1; // No pre-scaling
```

Some lines of code (25 and 34) are used to completely turn off the PWMs whilst they are being initialised.

```
SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 0;    // Disable TBCLK
EPwm2Regs.TBCTL.bit.CTRMODE        = TB_FREEZE; // Stop counter
```

3.3 Compares and Duty Cycle

Our ability to change the duty cycle of the PWM that we're creating is done by the compare registers (EPwmRegs.CMPA/B). In this case, we're only accessing the compare A, although another

compare is available to use. Using this compare value which is set to half of the time period we create our 50% duty cycle with lines 49 and 50.

```
// --- Set the compare values
EPwm2Regs.CMPA.half.CMPA = EPwm1Regs.TBPRD/2;      // 50% duty cycle

// --- Set actions
EPwm2Regs.AQCTLA.bit.CAU = AQ_CLEAR; // outA cleared on up count
EPwm2Regs.AQCTLA.bit.CAD = AQ_SET;   // outA set on down count
EPwm2Regs.AQCTLB.bit.CAU = AQ_CLEAR; // outB cleared on down count
EPwm2Regs.AQCTLB.bit.CAD = AQ_CLEAR; // outB cleared on up count
```

Recall I mentioned that this PWM is based off a triangle wave. Thus, there is an upward slope, and a downward slope. The upward slope counts from 0 to TBPRD=8000.0 and the downward slope counts from TBPRD=8000.0 to 0 and this repeats. What AQCTLA.bit.CAU says is, when the counter is equal to the value of Compare A (CMPA) and on the Upward slope, then clear the output value of EPWM2A (which is pin 27). AQCTLA.bit.CAD says when the counter is equal to the value of Compare A and on the Downward slope, the set the output value of EPWM2A.

The next two lines of code do the same thing but with output 2B (pin 23). See page 275 of the Technical Reference Manual for some clear diagrams and more example code of how this works.

3.4 Symbol Convention

I will mention here something about the way the register coding is done. When you set the value of a register, you're actually setting data bits. Therefore, you can only set 0 or 1, or 00, 01, 10, 11 etc. However, notice that you're setting the registers with words e.g. AQ_CLEAR. Well, select one of these words (actually called symbols) and right-click. Choose "Open Declaration".

This will take you to line 60 of one of the header files and you can see that actually, AQ_CLEAR is assigned the value of 0x1, which is the hex value for 01. If you now go to page 340 of the Technical Reference Manual, you will see that setting these register bits to 01 means to clear the output.

The reason AQ_CLEAR is used instead of 01 is because it is much easier to a programmer to understand what AQ_CLEAR is doing as opposed to two numbers. 01 means nothing, whereas AQ_CLEAR means the action qualifier is set to clear. Very useful!

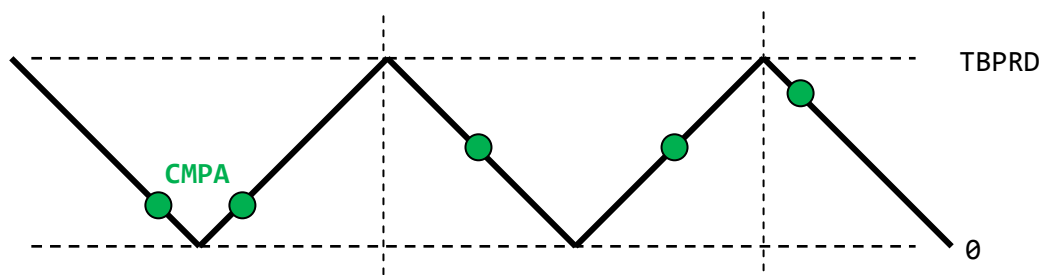
Now go back to the C-code file "Pwm_Init.c".

3.5 Shadow Registers

These are fairly minor components you have access to but worth noting. The shadow registers are used to control how your CMPA/B values change. For example, in one time period you could have

your CMPA value change multiple times and so may end up with some glitchy outputs of your PWM depending on your settings.

What the shadow registers do is they store in themselves any changes of the compare values. As a result, the current compare values remain unchanged in the active registers and only values in the shadow registers get modified. What line 57 then says is, when the compare counter is equal to the peak of the triangle wave (8000.0 in this case) then the compare values stored in the shadow registers can be loaded into the active registers



The above image tries to explain this. You can see that only after the peaks of the triangle wave does the CMPA value (represented by the green dot) move. Using the shadow registers provides a certain degree of stability.

3.6 Interrupts

Interrupts won't be covered until later on but there are a few lines of code covered here which show how a PWM can be used to generate an interrupt event. An event is what would possibly cause an interrupt to happen. So in this case, EPWM2 has been set up to cause an interrupt to occur every time the counter equal zero. Thus, the interrupt would be called at a frequency of 5kHz or every 200 μ s.

The last lines of code in this initialisation function simply enable the counting functionality of EPWM2.

4 PWM TASKS

4.1.1 Task ONE

Try debugging the code again now and running it. Carefully measure the signal at EPWM2A (pin 27) using an oscilloscope and probe. You'll probably want some jumper wires. What do you see?

You should see nothing. This is because you've not enabled the GPIO pins yet or the EPWM clocks!

Take a look inside the two C-code files DeviceInit.c and GpioInit.c and try and enable these things now. There should only be two things to change; one value in each file.

4.1.2 Task TWO

Now it's your turn to create some of your own PWM signals. All you have to do is the exact same thing in the example code but with some modifications. Try and do the following:

note: you should only need to use one compare (CMPA) for these tasks

1. Create a PWM signal from EPWM1 that has:
 - a 20kHz switching frequency
 - uses an up-down count
 - two outputs (A and B active) which switch opposite to each other
 - have a 25% duty cycle
 - have a dead-time of 20ns
 - shadow registers are enabled
 - interrupts are disabled
2. Create a PWM signal from EPWM3 that has:
 - a 100Hz switching frequency
 - uses an up-only count
 - has only one output active
 - has a duty cycle that changes in the polling loop every 2 seconds between 25% and 75% (think about how the LED toggle is timed and you'll probably need to add some custom C-code in the while loop)
 - shadow registers are not used
 - interrupts are disabled
3. Create a PWM signal from EPWM8 that has:
 - a 5kHz switching frequency
 - uses a down-count only
 - has a duty cycle of 50%
 - shadow registers are enabled
 - interrupts are enabled to happen on every 3rd event. Each event will occur when the counter reaches zero AND the period.

You won't actually be able to test the third task but it may be interesting for you to do. If you want have a look at the figure on page 160 and the table at the bottom of page 169 of the Technical Reference Manual and see if you can relate what's happening to some of the code lines you've just written (with regards to PIE groups, IER etc).

Remember to use the Technical Reference Manual to help you out and also, don't forget that when you "right-click -> Open Declaration" it can sometimes be very useful.