PWM in the Main Controller

Welcome back to Cypress Academy, PSoC 6 101. In the last video I showed you how to use a PWM to control the brightness of an LED. In this video let's put the PWM to work controlling my robot's servo motors.

We will add the PWM functions to our BLE controlled robotic arm project, which we called “MainController”.

The servo motor in my robotic arm requires a train of pulses with a frequency of 50Hz – which is also known as a period of 20ms - and the pulse width needs to be between one and two milliseconds. At one millisecond the servo motors are turned all the way to the left and at two milliseconds, they're turned all the way to the right. At one and a half milliseconds, it is right in the middle.

So, to make the math work, we’ll choose a 4MHz clock and pre-divide it by 4 which means that each of the PWM clock ticks will be 1uS. Then we'll set the period to 20,000 so we can go from 0 to 20,000uS which is also known as 20ms. For a 1ms wide pulse - also known as 1000 microseconds – I'll need a compare value of 1000. This also means that 1% will be 10 PWM clock ticks. Now I can set the compare value with software between 1,000 and 2,000 to give us full range of control over the motors.

I’ll start by dragging and dropping one PWM from the catalog on to the schematic. Let’s setup the period and compare to 20,000 and 1,000 respectively, turn on the divide by 4 pre-divider, and click OK.

We want to control two motors, so let’s copy that component and paste a second one into our design. It will be named PWM\_2 and that’s convenient since PWM\_1 will control motor 1 and PWM\_2 will control motor 2. The copy paste function is also nice in that all of the settings from PWM\_1 were copied over to the PWM\_,2 saving us some time.

Now let’s add two digital output pins to our design. I like the copy-paste function, so I’ll pull in one and rename it to “M\_1” for motor 1. A quick copy/paste and I now have “M\_2” as well.

Let’s connect those pins to the PWM output line on the components.

Next, we'll need a clock. I’ll drag and drop a clock in and set it to be 4MHz. Then wire it to both of the PWMs.

One of the most forgotten steps in using a PSoC Creator design is to assign the pins, so let’s do that now before we forget. Go to the pin settings and we’ll set M\_1 to P0[2] and M\_2 to be P5[5] which will match up to the H-bridge pins on the Arduino shield I’m using.

Let’s generate the application and start working on some firmware.

Create and edit the file pwmTask.h. Remember I said we are going to put all of these things into separate files so each of the tasks makes sense by themselves. Add the #prama once, and the includes for FreeRTOS and the semaphores. Next, I will create two enumerated types, one for the motors called motors\_t and one for position called motor\_pos\_t. These enums will be used in the structure called PWM\_Message\_t that the other tasks will use to communicate with our main PWM controller.

All right, let's define the structure to send messages to the PWM. This structure will allow me to tell the PWM task, that we’ll create here in a minute, what motor we want to change, whether we want a relative or an absolute change, and a percent change that we want to make. I’ll call this structure “PWM\_Message\_t”.

Finally, I will define the function prototypes for the motor task and a helper function called getMotorPercent that the other tasks can use to find out the current state – in other words the current position - of the motors.

After the pwmTask.h header file is built, we can go ahead and update the main\_cm4.c file. Four things need to happen:

1. Add an include for the pwmTask.h
2. Create a variable for the pwmQueue
3. Initialize the pwmQueue
4. Start the PWM Task

When you have different tasks communicating with each other via an RTOS queue, there's always a question of where you define and initialize the queue. I like the main function to “own” these global-ish variables and then I like to tell the other files about them in a file called global.h.

So, create global.h, add the #pragma once, put in includes for FreeRTOS and semaphores, define the extern for the pwmQueue. Now the other files can get access to the queue just by including global.h.

Now create and edit the PWM task (pwmTask.c).

The only purpose of the pwmTask is to take a message from the other tasks in the form of a percent - meaning a number between 0 and 100 - and then turn the percent into a pulse width between 1ms and 2ms. And finally turn the pulse width into a compare value which can be written into the correct PWM compare value register so that you get an output pulse of the correct frequency and width.

I am going to want 3 pairs of functions:

1. Percent to pulse width and pulse width to percent
2. PWM compare value to pulse width and pulse width to PWM compare value
3. Percent to compare value and compare value to percent

So, how do I do that? Well, the math is simple. Look at this graph. On the X-axis I have a pulse width in micro seconds and on the Y-axis I have percent. When you have a pulse width of 1000 us you should have a percent of 0…here is the first point of the line. Then when you have a pulse width of 2000 us you will have 100%, that is the next point on the line.

From algebra one we remember y = mx + b… OK, so the formula that turns pulse widths into percent is percent = slope times the pulse width minus 100 percent. The slope of the line is rise over run, or 100 percent minus 0 percent over 2000 microseconds minus 1000 microseconds.

And that can be arranged to percent into pulse width like this:

Pulse width = percent plus 100 times the slope which is 2000us minus 1000us divided by 100 minus 0.

Or, simplifying further, pulse width = percent times 10 plus 1000us.

So, let's start this thing by creating the helper functions. First define the ranges of our pulses… MIN\_US and MAX\_US. Then define the PWM parameters PWM clock, and PWM divider. These could be read from the schematic… but I wanted to make this simpler to understand so I'm hard coding them here. Next derive the number of PWM ticks per microsecond.

Now I can build the helper functions.

Percent to pulse just takes a percent and turns it into a pulse width with the formula I derived earlier… then pulse to percent does the inverse of that.

The two helper functions to turn compare values into pulse widths and pulse widths into compare values. These two functions use the settings on our schematic to figure this out.

And finally, two functions to turn compare values into percent and percent into compare values.

Now in my program I can use these helper functions to set the PWMs using percent or find out what percent is currently set by reading the compare values of the PWMs.

The next function I build will be used by other tasks to look at the current state of the PWMs in percent. I'll call this function getMotorPercent. The getMotorPercent function takes as its input either M1 or M2, then it looks at the PWMs and figures out what compare value is being used by that PWM right now. Then it uses the helper function compare to percent and returns the current percent value of that motor.

Now I’ll create the PWMTask. When the task starts I’ll start the two PWMs using the start API; then in the infinite loop I’ll wait to receive an RTOS command from the PWM queue. When I get a command from the queue, I’ll figure out what the hardware and counter numbers are so I can use the appropriate macros.

Then what I’ll do is, if the message coming in wants to make a relative change in percent, I’ll get the current compare value, convert it to percent, and make the change. If the message calls for an absolute change, I’ll make the change directly. Then I’ll update the compare value of the appropriate PWM.

That’s it for the PWM Task.

Now, I need a way to test this, so I’m going to add it to the UART command set that we defined in an earlier video. In UARTTask.c I need to add includes for global.h and pwmTask.h. And down in the UART Task I’m going to use the commands “o” and “p” to change the relative percent value by negative 10% and positive 10%, respectively. And I’ll use the commands “k” and “l” to do the same thing for motor 2. In the case statement for the command ‘p’, I’ll set the message to control motor one, I'll set an absolute percent, and then I'll set the percent change to 10. Then queue the message and break. Now I’ll do the same for the other commands.

In order to make the debugging easier I am going to add a command to printout the current status of the motors. So, add to the case statement for the ‘s’ that printf's the getMotorPercent for M1 and M2.

Are we done? No! We can’t forget to add the messages to the help command. So, I’ll do that real quick.

And that’s it…that’s the beauty of an RTOS. Simplifying complex designs.

Let’s build, program and test…

Now we have a functioning UART controlled robotic arm, but we’re not done yet. This is supposed to be a BLE controlled robotic arm. So, we still have a little more work to go. In the next video, I will walk you through setting up an I2C control interface.

You can post your comments and questions in our PSoC 6 community or as always you are welcome to email me at alan\_hawse@cypress.com or tweet me @askioexpert with your comments, suggestions, criticisms and questions. Thank you.