I2C

Welcome back to Cypress Academy, PSoC 6 101. In this video, I will show you how to setup an I2C dash board for our BLE-controlled robotic arm project. An I2C control and debug interface is handy to understand because you will more than likely have multiple sensors or ICs in a typical system that will require this type of interface. Let’s start first by learning about the I2C peripheral and the EZ-I2C component.

Let’s start with a new project. I’ll call it “BasicI2C”.

Again, I love copying and pasting as it saves so much time, so let’s go back to the schematic view of the BasicTCPWM project we did in the prior lesson. Let’s copy the entire schematic and paste that into our BasicI2C project.

I want to show you an additional feature of the TCPWM component, so let’s open up the configuration dialog. Click on the advanced tab. Here you can see more configuration options like the kill and start input, which we’ll talk about in the next lesson, as well as how you can change the polarity of the PWM output. In this case, because the LED on the kit is active low, and I want to use larger duty cycles for a brighter LED, cause that just makes sense, I’m going to invert the output of the PWM. Now hit okay.

Now, let’s add the EZ-I2C component from the catalog. Double click it and let’s rename it to EZI2C without the underscore 1. You can also see the other configuration settings like the address and data rate. Click okay and now on to configuring the pins.

And let’s make sure we assign the pin for the LED to P0[3]. And then SDA, the data signal, assign that to P6[1] and SCL, the clock, to P6[0].

And on to the Cortex-M4 main application.

Let’s start the PWM and EZ-I2C components with the simple start API call.

The EZ-I2C component implements an EEPROM I2C scheme. It’s interrupt driven and designed to easily setup an I2C slave in your firmware.

We’ll initialize a read/write buffer with the set buffer API call and a local variable that we’ll declare called myBuffer. Because this variable can be changed by the interrupt service routine I’ll use the volatile flag to let the compiler know not to optimize the variable out.

In the main loop, we’ll simply use the contents of the variable mybuffer to update the compare value of the PWM using the standard API call that we used in the previous BasicTCPWM project.

Then, we’ll put the CPU to sleep and tell it to wait for the next interrupt before waking up.

Now, build, program and test…

To test this I’ll use Cypress’ bridge control panel tool that comes with PSoC Creator. I’ll open the tool and click on the KitProg entry at the bottom of the screen. At the top I’m going to write different compare values to send to the PSoC 6 to change the LED intensity. I’ll do this by typing: w, for write, 8, for the address, 0, for the register, and the hex value for the intensity, 0 to 100; and then p, for an I2C stop. For example, w 8 0 32 p, sets the PWM at a 50% duty cycle for half intensity.

Awesome.

[Alternate 2-video version]

Now we that we understand how to implement I2C with PSoC 6, next we’ll add this to our BLE-controlled robotic arm project.

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 at @askioexpert with your comments, suggestions, criticisms and questions.

Welcome back to Cypress Academy, PSoC 6 101. In this video, I will show you how to add the I2C interface to our BLE-controlled robotic arm project. I’m going to use the bridge control panel as a dashboard display to show the position of the motors. So, as we change the position of the motors you’ll see a graph update in the bridge control panel showing where they are. Sound cool?

[1-video version]

Now, let’s add this into the BLE-Controlled Robotic arm project. I’m going to use the bridge control panel as a dashboard display to show the position of the motors. So, as we change the position of the motors you’ll see a graph update in the bridge control panel showing where they are. Sound cool?

[Merge]

Open the BLE-Controlled Robotic arm project’s schematic.

I’ll add the EZ-I2C component from the catalog. Change its name to EZ-I2C and leave the rest as default.

Assign the pins to P6[1] and P6[0] for SDA and SCL.

Generate the application.

Then we create and edit the ezi2ctask.h. This file will just have #pragma once and definition of the ezi2ctask.

Once that is done we can edit the CM4 main application. When the whole main controller is done there will be a two ways to display updates the position of the motors specifically the ezi2c and the capsense and the ble. In order for everyone to know that there has been a change to the pwms I am going to use an event group. An event group is essentially an RTOS safe global uint32 variable. All of the tasks can read the status of an event group without fear of a race condition. They can also wait for a change in the event group. This is how we are going to communicate from the PWM task to the ezi2ctask.

Lets get to editing in main\_cm4.c. First include ezi2ctask.h and the freertos event\_groupss.h. Next define the event group pwmEventGroup. And in the main function ill initialize the pwm event group by calling xeventgroupcreate and start the ezi2c task.

Before anyone can use the pwm event group I need to add it to global.h. In this file Ill add the include for event\_groups, add an extern for the pwmEventGroup and finally create a definition of the event group. Basically a bit mask. The first bit will be for the I2c and the second bit will be for the BLE.

In the pwmTask.c all I need to do is set all of the bits when the PWM has changed using the xevengroupsetbits RTOS command.

Now let’s create and edit the file EZI2C task.c. This is a fairly straight forward file. First include the project dot h, the freertos dot h the pwmtask dot h and finally the pwmtask dot h. Next make the function ezi2ctask … I’ll setup a variable to store the percent value of the motors so the I2C master, the bridge control panel in this case, can use to read and display that data. I’ll call it motorpercent, an array of two unsigned 8-bit integers for the two motors. I’ll start the EZI2C component. Setup the buffer and mark it as read only. Start the infinite loop and then initialize the motorpercent array with the current value of each motor’s PWM compare value. Then I’ll wait for a change in the event group, and do it again.

What happens is this task will go to sleep until the pwmTask changes the event group. When that happens the task wakes ups, then updates the values in the buffer, then waits for another change.

That’s it! Slick isn’t it?

Now build, program and test. Note if you still have the bridge control panel open and connected to the kit, you need to disconnect the kit from the software by clicking on the disconnect icon in the bridge control panel. Then go back to PSoC Creator and hit program again.

Now, go back to the bridge control panel and connect to the kit.

Let’s setup the chart so we can graph the motor percent values. Go to the chart menu, select variable settings. For the first row, click active and type M1 as the variable name; second row, click active and M2. Okay.

Next in the editor I’ll type w 8 0 r 8 @M1 @M2 p; What this does is sets the read pointer to zero and then reads in two bytes which it stores in M1 and M2. Now press enter a few times to see it work.

Then go to the chart tab.

And hit repeat.

Now you can change the values via the UART interface by using the keys o, p, k and l in a terminal client that we setup before.

Success!

Now we have an I2C dashboard interface for our robotic arm project. In the next video, I will walk you through how to add custom digital logic to implement the Terminator-kill, or safety-switch to the design that will safely stop the motors.

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 at @askioexpert with your comments, suggestions, criticisms and questions.