Hello World!

Welcome back to Cypress Academy, PSoC 6 101. In this video I will show you how to build your first set of PSoC 6 projects.

As with any first C program, we’re going to start off with a Hello World, blinky LED, but with a dual-core twist.

Let’s pause here for a moment and talk about the system architecture briefly. The PSoC 6 BLE device we are using is a dual-core ARM Cortex-M4 and Cortex-M0+. This is a shared memory, multi-processor architecture, meaning both processors can talk to everything in the chip, RAM, Flash, peripherals, etc. With the advanced security features of PSoC 6, which we’ll talk about later, provide you the ability to lock down your project in a very flexible way. Security is central to the identity to this chip. So, it would be good to think about what role each of these cores is going to play in our design. The Cortex-M0+ core is, of course, the lower power processor versus the Cortex-M4…so, it would be good to dedicate the Cortex-M0+ core for the tasks that require the core to be in an active power mode for longer durations and vice versa for the Cortex-M4.

For this example, we’re just blinking an LED so we’re going to blink it in three different ways: 1) hardware (no CPU); 2) Cortex-M0+ firmware; and Cortex-M4 firmware.

For our BLE-controlled robotic arm project, however, we’re going to dedicate the Cortex-M0+ core to our BLE connectivity tasks as well as security tasks that we’ll talk about in a future lesson. We will use the Cortex-M4 core for the rest of the application, driving the servo motor PWM signals, reading the digital sensor values from the motion sensor, outputting data to the E-ink display shield, etc. The architecture is very configurable so, depending on your application and needs, this might differ but I will show you how to work with each core independently so you can customize it on your own later. But let’s get back to blinking the LEDs by picking up where we left off last time with the Hello World PWM project.

Let’s start with blinking the LED with no CPU control.

To do that, let’s start by placing a digital output pin component by dragging and dropping into the schematic. Double click to configure it. Change it’s name to red—for the red LED obviously.

The next thing we’ll do is get a TCPWM Component and drag it in. Double click that component. Change it’s name to PWM. We’ll change the period to 1,000 and compare to 500 for a 50% duty cycle. We’ll draw a wire from the PWM output to the LED.

Now we need a clock to drive the PWM component. Grab a clock component from the catalog and drag that in. Double click the clock to change the frequency of the clock to 1KHz.

Now, we need to connect the red LED pin component to the actual pin on the PSoC 6 device. So, in the design wide resources folder, double click Pins. We’ll assign the pin to P0[3].

Now let’s generate the project firmware.

Now let’s edit the CM0+ main application. I’ll comment the CY\_SysEnableCM4 to keep the Cortex-M4 off, start the PWM by calling a simple start API and then force the CM0 to go to sleep by adding a system API call CY\_SYS\_PM\_Sleep…

And that’s it, we’re ready to build and program. Click the program button and it will build the application and program the device that I have already connected to a USB port on my laptop.

And tada, it’s working!

Now, this time let’s have the Cortex-M0+ core control the LED.

Let’s start by creating a new project in our workspace, but let’s call it HelloWorldCM0p for the M0+.

Like last time, let’s drag and drop a pin component and configure it. Let’s call it red again. This time let’s turn off the hardware connection because we’re going to control it by firmware.

Next we need to assign the red pin component to a physical pin, P0[3] again.

Now let’s generate the application.

And now to the main c application file for the CM0+.

Once again, we’ll comment out the CM4 enable line.

I’m going to use the new peripheral driver library to control the state of the pin component. Let me show you an example. Go to help->PDL, under the API reference, under drivers, under GPIO, under functions, under GPIO functions, you’ll find all of the different functions to interact with pins. The one we’re going to use is CY\_GPIO\_Write.

Let me show you how to use this. In the main application, add the PDL API, CY\_GPIO\_Write. We need to send which port we’re writing to so we can use a macro that’s created by the pin component we added to the schematic called “RED\_PORT”, and the pin number called “RED\_NUM”, and the state either high or low, 1 or 0. Then let’s add a delay using the CyDelay API and some number of milliseconds—we’ll use 500 milliseconds. We need to then change the state again, this time 0. And add another delay.

That’s it…now let’s build, program and test it.

Lastly, this time let’s have the Cortex-M4 core control the LED.

This time we’re going to go to the workspace explorer and copy and paste the project we just created. I’m going to right click->copy and right click->paste. Now let’s rename this project by right clicking the project name and selecting rename. Let’s call it “HelloWorldCM4”.

Now let’s start in the CM0+ main application file. Uncomment the enable CM4 line so we can turn on the CM4 core. And now select and cut the code that we added before and let’s paste that into the CM4’s main application file.

That’s it…now let’s build, program and test it.

What I just showed you is that the GPIO is accessible from both cores—that means the registers are connected to the same bus as the cores are. Meaning, as the application builder you have the flexibility to partition your application to your needs.

If you’ve used a PSoC 4 before you’ll remember we create a customized set of APIs for each instance of a component, meaning if you have multiple instances then you may have multiple sets of the same firmware and APIs in your code. We have moved to a more standard model with PSoC 6 such that we have a driver for each type of peripheral and you specify which instance of a peripheral you talk to in the API call. That is our new Peripheral Driver Library, or PDL for short. But the cool, PSoC Creator twist, we create helpful macros for you like RED\_PORT and RED\_NUM in the previous projects to simplify your firmware development.

Now that you understand the basics of the environment and how to create a project, in the next video I will walk you through setting up a FreeRTOS environment.

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.