Hello World!

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

As with any first time C program, we’re going to start off with the Hello World, and in the embedded world, the Hello World is also known as the Blinky LED, but this time I'm going to throw in a dual-core twist.

Let’s pause here for a moment and talk about the system architecture briefly. The PSoC 6 BLE device that we are using for these videos is a dual-core ARM Cortex-M4 and an ARM Cortex-M0+. This is a shared memory, multi-processor architecture, meaning both of the processors can talk to everything in the chip. That includes the RAM, the flash, and the other peripherals. With the advanced security features of PSoC 6, which I’ll talk about in later videos, we provide you the ability to lock down your project in a very flexible, very secure way.

The Cortex-M0+ core is, of course, the lower power processor versus the Cortex-M4 which is much more powerful and you can do a lot more with but you pay in power. In general, it's good to dedicate the Cortex-M0+ core to the tasks that require the core to be in an active power mode for a longer duration and vice versa for the Cortex-M4.

For this example, we’re just doing a blinking an LED so I'm going to show you how to blink it three different ways: first of all just using the hardware in the chip - no CPU involvement; the second way using the Cortex-M0+ and building firmware that works on the M0+; and finally on the Cortex-M4 and using firmware that runs inside of the Cortex-M4.

For our BLE-controlled robotic arm project, however, I'm going to dedicate the Cortex-M0+ core to be the BLE connectivity tasks as well as the security tasks that I’ll teach you about in a future video. We'll use the Cortex-M4 core for the rest of the application, and that means driving the servo motor PWM signals, it means reading from the digital sensor like the motion sensor, and it includes outputting the data to the E-ink display. This architecture is very configurable so depending on your architecture needs and your application problems, your implementation may differ slightly but I'm going to show you how to work with each core independently, so you can customize this thing to your own needs later.

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 by blinking the LED with no CPU control. We'll just use the PWM that's inside of the PSoC 6.

Let's start the project by placing a digital output pin from the component catalog by dragging it and dropping it into the schematic. Double-click to configure it. Change its name to RED – obviously the RED means it's the red LED.

The next thing I'll do is I'll get a TCPWM component and I'll drag it into the schematic. I'll double-click that component, change its name to PWM. I’ll change the period to 999 and the compare to 500 which will give me a 50% duty cycle because the counter will count from 0 to 999. 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 it into your schematic. Let’s double-click the clock and change its frequency to 1KHz.

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

Now let’s generate the project firmware by selecting Generate Application.

First let’s edit the CM0+ main application. I’ll comment out the section that says CY\_SysEnableCM4 and that will keep the CM4 from ever being turned on. I'll start the PWM by calling the simple start API and then I'll force the CM0 to go to sleep by adding the API call CY\_SYS\_PM\_Sleep…

That’s it. Now we’re ready to build and program the kit. Click the program button and it will build the application and program the device that I have already connected to the Type-C port on my laptop.

When you click on Program, you may see a Select Debug Target window. If so, you need to select either the CM0+ or the CM4 to specify which device's SWD interface to use since they can both connect to the debugger. It doesn't matter which one you choose since they are both able to program the flash with your project, but since it is more likely that you will want to debug the CM4 later, I'll choose that one.

Then I click on OK/Connect and the device is programmed for me. And ta-da! It’s working!

Now, 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. I'll call it RED again, but this time I'm going to turn off the hardware connection because I'm going to drive it with the firmware.

Next, we need to assign the red pin component to a physical pin. You do that by double-clicking the Pins inside of the Design Wide Resources and selecting P0[3] for the LED.

Now let’s generate the application.

In the main C application file for the CM0+, once again I’ll comment out the CM4 enable line because we don't need the M4 to turn on.

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, and then under the API reference, under drivers, under GPIO, under functions, under GPIO functions, you’ll find all of the different functions that allow you to interact with the pins. The one that I'm going to use is CY\_GPIO\_Write.

Let me show you how you use this. In the main application, add the PDL API CY\_GPIO\_Write. We'll need to tell it which port we’re going to write to, so we're going to use the macro that’s created by the pin component we added to the schematic called “RED\_PORT”, and the pin number called “RED\_NUM”, and then state either high or low, meaning 1 or 0. Then let’s add a little bit of delay using the CyDelay API and some number of milliseconds – I'll pick 500. We need to then change the state again, this time to 0, and the we'll add another delay. That’s it…now let’s build, and program and test the kit.

Finally, this time we'll make another project, but this project will run on the Cortex-M4 core, and it will control the LED.

So, instead of making a new project, let's go to the Workspace Explorer and copy/paste the project we just created. I’m going to right-click on the project name and select Copy. Then right click on the workspace name and select Paste. Now let’s rename the project by right-clicking the project name and selecting Rename. Let’s call it “HelloWorldCM4”.

In the CM0+ main application file, uncomment the enable CM4 line so this will turn on the M4 when the chip boots. Now let’s select and cut the code that we had originally in the CM0 file and then we'll paste it 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 GPIOs are accessible from both of the cores – the M0+ and the M4. That means the registers that control the GPIOs are connected to the same bus as the cores. This means, as an application builder you have the flexibility to partition your firmware into which core best meets your needs.

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

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

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