BLE Intro – CySmart to Robotic arm

Welcome back to Cypress Academy, PSoC 6 101. In the chapter 3 videos, I will show you how to add BLE connectivity to the robotic arm project that we’ve been working on. That will include both adding BLE to the MainController as well as building a remote control. Don’t worry… we’ll start simple and then build up from there, so let’s get started.

For this lesson, you will need either a BLE-enabled smart phone and an app from Cypress called CySmart, or the PC version of CySmart as well as the CySmart BLE USB Dongle that’s included in your PSoC 6 BLE Pioneer Kit. Whichever version of the CySmart application you choose will be used to read and write BLE control signals to the PSoC 6 BLE robotic arm.

In this video I'll build the equivalent of the hello world project… actually, the blinking LED project… for Bluetooth Low Energy. That example is also known as the Find Me project.

Let's start by creating a project called 3-1-BLEFindMe. This project will have one task for BLE which we'll call “bleTask”. It will manage … get this … the BLE. And it will run all of the required stuff to be an immediate alert service or IAS. The immediate alert service is specified by the Bluetooth special interest group and is a simple peripheral that you can tell to be in High Alert, Mild Alert or No Alert. Think of a tag that you could attach to your car keys. When you lose them, you could get on your phone and set the tag into one of the three states. For example, it could beep and flash in High Alert, it could just flash in Mild Alert, and it could be totally off in the No Alert state.

I'm going to use printf and FreeRTOS, so, change the build settings to enable both of those libraries … just like almost every chapter so far.

In FreeRTOS.h, turn off the warning, increase the heap size, enable semaphores and modify the syscall priority.

Once you've done that, let's edit the schematic. First add the BLE component, then add a UART component, then four digital output pins: one for LED9 (to indicate a connection), one for the red, green and blue LEDs to represent the three alert levels – none, mild and high.

Go ahead and change the name of the digital output pins to led9, red, and blue and set their initial state to high … also known as off. Then set the last pin to be named green and set its output state to low … which will cause the green LED to be on by default indicating the initial state of no alert.

Once you have that done let's go ahead and assign the pins in the DWR… first the UART to be P5[0] and P5[1] then blue is P11[1], green is P1[1], led9 is P13[7] and finally the red is P0[3].

Now we need to configure the BLE. This device is going to be a “GAP Peripheral” that accepts one connection. As I have talked about earlier, one of the cool things about PSoC 6 is that it has two cores… remember a CM4 and a CM0+. Our BLE implementation will let you run the controller part of the BLE stack on the CM0+ and the rest of the BLE stack on the CM4. To do this you just need to select “dual core”.

Once you've done that… let's edit the GATT Settings. The immediate alert service is specified by the Bluetooth SIG. We made it easy for you to add into your project. Just right click on the “server” and select add service… then immediate alert.

Now that our GATT database is setup, we can configure the GAP settings. First, let's name this device “FindMe”… then setup the advertising settings… let's pick general discovery mode, and no timeout on the advertising…

Now let's configure the advertising packet to have the name of the device as well as the service UUID of the alert service.

That's it for the schematic. Now you need to run Build Application to bring in all of the middleware and setup all of the connections.

In order to make this work you are going to do something new… that is, you are going to edit the cm0p.c. In this file, I need to start the BLE by calling Cy\_BLE\_Start… then in the main loop I need to call Cy\_BLE\_ProcessEvents.

With all of that done we are now ready to edit the main\_cm4.c to have the main loop of the BLE firmware.

First let's setup the includes: project.h, stdio.h, Freertos.h, task.h, semphr.h and limits.h. The limits.h is included so I can get the UINT\_MAX.

The LEDs on the board are active low… so I'll setup macros for them. LED\_ON is a 0 … and LED\_OFF is a 1.

Then we need a semaphore for the BLE interrupt called the bleSemaphore and a taskHandle for the BLE Task.

The way that our Bluetooth stack works is that you provide a function which can process events. We call these functions “callbacks”. When something happens that we think is interesting we'll call your function with a parameter that will tell you what event occurred, and then, in that function, you will need to do the right thing.

For this project we'll need to create two callbacks. One for handling the generic Bluetooth events and one for handling the immediate alert specific events. First let's tackle the generic event handler.

I'll declare the callback function …. It will return a void … meaning nothing … I'll call it genericEventHandler … that makes sense … the Bluetooth stack will pass it an event code which is just a uint32\_t and it will give it a void pointer to an event parameter.

All of these event handler functions look about the same. They're just a big switch statement that look at the different kinds of events that can occur in the system. For this example, there are only three events that we care about: when the Stack turns on; when a device is connected (like your phone); and finally when a device is disconnected.

First, when the stack turns on, also known as CY\_BLE\_EVENT\_STACK\_ON. For this case all I want to do it printout a message that the stack has turned on… and then I want to start advertising. When I call the start advertisement function, our device will start sending out the advertising packets over the radio. This will let other devices see that we are there, it will tell them our name and will tell them that we have an immediate alert service. Remember earlier when we specified all of this information in the BLE customizer on the GAP Settings page.

The next event is the device connected event. When that happens I'll printout a message that we are connected and then turn on led9 to indicate that we have that connection

The last of the generic events is the disconnect event. When that happens I'll printout a disconnect message, turn off led9 and then tell the BLE to start advertising again.

That all for the generic events.

The next function is the event handler for the immediate alert service. Just like the last handler it returns a void … and I'll call it iasEventHandler… the BLE stack passes it an event code and an event parameter.

First, I will declare a uint8\_t called the alertLevel that I will use to temporarily hold the alert level.

Then I will see what event has been sent to me. It turns out there's only one possible event, specifically CY\_BLE\_EVT\_IASS\_WRITE\_CHAR\_CMD. In other words, the only event that can occur is that the other side… the phone side… the GAP central side… sent me a new value for the alert. I built this function so that if one day in the future there's a new event created I can easily add the ability to handle that into this function.

So, I will call the GetCharacteristicValue function to find out what value the phone sent me. I tell this function that I want the alert level and I tell it to store it in the alert level variable which is just one-byte long.

Once I know the alert level I can decide what to do with it using a switch statement. There are three possible alert values: none, mild and high… all I do is printout the message and then turn on the correct LED.

Hey, that's pretty simple.

The next step is to build an interrupt service routine. This function will be called every time you need to call the process events function. The problem is you don’t want to call the process events function inside of an interrupt service routine. So, what are you going to do? Simple. You are going to give a semaphore which will indicate to your main loop that it needs to call the process events function. Now let's look at the ISR. All it does is give the semaphore, and possibly yield to a higher priority task.

You could have skipped all of this ISR business and had the infinite loop repeatedly calls the process events function – like we did in the CM0p file. The problem with that is most of the time it wouldn’t do anything and as such would be a complete waste of CPU time. Really, we'd rather only call this when there's something to do.

The next part of this system is the main bleTask. The task is responsible for initializing the BLE, registering the callbacks, and then running at the right time and calling the process events function.

So, declare the task, then printout a message that the BLE task has started, initialize the semaphore that will be used in the ISR, then start the BLE stack and register the generic event handler callback function.

Now you need to get the stack started. This requires a number of events to be processed… so we'll just have that in a while loop that runs until all of the process events have happened.

Once the stack is on we can register that we want callbacks when something needs to be done by giving the ISR, and we want to be called back when the immediate alert service is written, so we register that callback function as well.

Finally, we build the infinite loop. This loop will wait until the semaphore is set in the ISR, then it will process events. This is cool, because the task will go to sleep until there is something that needs to be done.

That's it for the BLE task.

The last part of the firmware is the main function. All that needs to happen here is we need to initialize the UART, create the BLE task and start the RTOS scheduler.

Now we can build, program, and test this thing.

When you program the kit, the first thing you see is the green light turn on. That's good. Now you can start CySmart on your phone… pull down to refresh the list of devices… where I am standing right now I can see a bunch of different Bluetooth devices… but the one that I am interested in is called “FindMe”. Remember we gave that name in the GAP settings in the Bluetooth customizer.

When I click on FindMe, you see that led9 turns on indicating that we have a BLE connection. That good. Now I can swipe over to the FindMe profile in CySmart. And when I click on that it gives me the option of setting mild alert… hey look the LED is blue… that’s good. Then high alert… and yes, the red LED turns on… and if I go back to no alert it goes back to green.

Alright, all that seems to work…

Finally, I click back to disconnect and as soon as I do, led9 turns off indicating that I no longer have a Bluetooth connection.

That’s it.

In the next video I'm going to show you how to build a custom BLE service for CapSense that we will then integrate into our MainController for the robot.

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