BLE Simple Central

Welcome back to Cypress Academy, PSoC 6 101. In the previous lessons I showed you how to build BLE peripherals… specifically a peripheral running the immediate alert service and a peripheral running a custom service. In those lessons we used your cell phone to talk to those devices. In this lesson I'm going to show you how to create the other side of the connection. The “other side” is more properly called the central… or more specifically the GAP central.

The way that a BLE central works is it starts by listening for advertising packets… this process is called scanning. When it hears a peripheral that is advertising something interesting it can then initiate a connection to that device. After the connection is made, it can discover all of the services on that device and after that's done it can read and write the characteristics on the remote device.

You might have guessed from our previous lessons that there will be events that are called back to you each time to indicate that these steps have happened, and you'd be exactly right.

With this project I'm going to show you how to take care of each of these steps that I described earlier, and I'm going to show you how these events happen and when. You will then be able to program your development kit to see all of these things going on in BLE land.

For this lesson we will build a central that will look for the red LED brightness peripheral that we programmed in lesson 3-2. It would be better if you had two development kits, one programmed with the LED brightness peripheral firmware and one to do this lesson. If you don’t, then by all means order one up from one of our partners…Mouser or DigiKey, or Arrow, or a bunch of different place. Or you can just follow along and learn with me.

This project will scan the BLE airwaves looking for a BLE peripheral that is advertising the LED brightness service, it will then connect to it, discover its services, and then it will use the + and – keys on the UART to change the brightness of the LED.

Alright, let's get started. To make things easier we're first going to open the simple BLE project and save the LED service so that we can use it again in this project. That way we don't have to type in all that UUID stuff. Go to the schematic, double click the BLE, click GATT settings, the click on the LED service. Next, press the little disk image and select “save profile…”, then save the profile as LED.service to somewhere you can find it again (like say on your desktop).

Now close that project and create a new project called SimpleBLECentral. First let's edit the schematic. Add the BLE Component, add a UART, and a digital output pin which we will call LED9. Remember, I like to use LED9 to indicate the presence or the absence of a BLE connection. Go ahead and change its name to LED9, make the initial state high – so that it begins the process OFF and turn off the hardware connection.

Now edit the BLE component. This time instead of making a BLE Peripheral, I will select BLE central. Once again, I'll run this project in dual core mode.

Next, you'll edit the GATT Settings. In order to make the service discovery process work better, PSoC Creator gives you the ability to tell the BLE stack what services you're looking for during discovery. This is called a client profile. Unfortunately, we didn’t give you the ability to add a custom service. So, you need to do a little work around. Right click on GATT and select add profile… then Find Me… then Find Me Locator GATT client. Next, right click on the Immediate Alert and pick delete. Now that we have a blank client profile, the next step is to load the LED service that we saved into the file earlier. To do this right click on the client and say Add Service From File. Then pick the file LED.service.

When you expand the LED service you should recognize all the stuff we setup earlier.

Next you can go to the GAP settings. First give this device a name, and then change the scan settings to always do fast scan… I ain't got time to wait boys and girls so I want to see the fast mode! Now apply that.

I almost forget to assign the pins and I know Greg would laugh at me if I didn't get that done. So, go to the DWR and assign the UART to P5[0] and P5[1] and the LED9 to P13[7].

As I have in all of our projects I am going to use FreeRTOS and the Retarget I/O library. So, go to the build settings and include those into the project.

Now hit generate application so that PSoC Creator can do its magic.

Next, I'll update the stido\_user.h and tell it to include the project.h and use the UART\_1\_HW as the standard in and standard out.

In the FreeRTOS configuration, get rid of the warning, turn on semaphores, make a bigger heap, and change the MAX\_SYSCALL\_INTERRUPT\_PRIORITY just like we did in the prior BLE projects.

The next setup step and fix the main\_cm0p.c so that it runs the BLE controller… so turn on the BLE stack… then in the infinite loop process events like, yeah, like that.

Finally, we're ready to do some programming… my favorite part.

Open main\_cm4.c

This whole program will have three interesting functions. First, writeLed, which will take a value as an input. Then if you are connected it will write that value into the BLE peripheral that you are connected to. So, let's write this function.

Let's see… returns a void… function name is, oh, writeLED … takes a unit8\_t as a parameter… which we'll call brightness.

Now let's ask the BLE if we're connected and have completed the service discovery… in other words we are connected and we know the BLE handle of the green LED brightness characteristic. If that isn’t true… then just print a message saying no way… and return.

So now we know we are connected… printout that we are writing to the brightness.

In order to send a BLE write you need to call the function Cy\_BLE\_GATTC\_WriteCharacersticValue. GATTC means GATT Client… In order to call that function we need to have a structure of type cy\_stc\_ble\_gattc\_write\_req\_t … and, yes, that is a mouthful. Once again, all the guys in this studio watching me blab that out painfully as we had to do it 3 or 4 times.

That structure contains all of the information about the characteristic that I want to write to.

Each characteristic in the GATT server which is running on the GAP peripheral is identified by a 128 bit UUID… and that's a bunch of bytes…actually 16. Rather than send 16 bytes in every BLE transaction, we will create a 1-byte alias. That 1-bye alias is called the handle. The mapping of handles to UUIDs is sorted out during the service discovery process which I will tell you more about in just a few minutes.

The bottom line is that for us to write the LED brightness characteristic we need to specify the correct handle and it turns out that the Cypress BLE stack figures that out for you and stores it in this crazy array. This is the first thing that we have to save in our structure.

Next, we need to give a pointer to the value… we need to tell it how long it is… yeah, uint8…1 byte… those both go in our structure next. Then we need to tell it which connection. Remember when we set things up we configured only for one connection at a time, so I hardcoded the connection. Now, that's the last bit of information that our structure needs.

Once all of that is setup I can finally call the write function and if it doesn’t work I just printout an error.

The next function we will write is called findAdvInfo. This function parses through a BLE advertising packet and looks for the name and the service UUID. The format of these advertising packets is specified by the Bluetooth SIG… but it's pretty simple. When you get an advertising packet you know its total length. The packet is then divided up into a variable number of fields. The first byte is the length of the first field. The second byte is the type of field, then the next length minus 1 bytes is the data in the field. This means you can scan through the packet and look at each of the fields to find out what you're looking for.

Luckily, the Cypress BLE component has a cool tool for looking at advertising packets. Let me show the advertising packet that you specified in the Simple BLE Peripheral. First, open the project… then open the schematic… then double click on the BLE… then go to the GAP settings… and finally to the advertising packet. This is cool. You can see the total length of the packet is 28 bytes. It has three fields in it… the first field is 3 bytes long, the second is 7… and the last is 18 bytes long. On this tool you can see the other types of fields. All of these fields are specified by the Bluetooth SIG, but I'm really only interested in the name field, which you can see right here is 0x09 and the service UUID field which is 0x07.

Now that we know the advertising packet format, let's go write a parser. First, I'm going to store the name and name length, and UUID and UUID length into a structure… so let's declare that structure.

Then I'm going to make a function which will return void and will take pointer to an array of data of type uint8\_t… those are just the raw bytes of the advertising packet… and the length of the packet.

First, I'll zero out my structure.

Then I'll use a for loop to look through the packet. The first byte… aka adv[i] is the length… and adv[i+1] is the field type. So, look at the first field type… if it is 0x07 or 0x09 then save the information about it… otherwise jump to the next field.

Cool. When I get an advertising packet now I can just look for the names and the service UUIDs.

All right… the next function is the event handler. The function just looks like every other event handler that we've already built.

If I get a stack on or a disconnected, then I want to turn off the connection LED and start scanning again. What this will do is tell the BLE radio to start listening for advertising packets and when it hears one it will call the event CY\_BLE\_EVT\_GAPC\_SCAN\_PROGRESS\_RESULT to tell us that its heard another device.

So, let's deal with that event. On the UART I will printout that I heard the device, then I'll print out the BD address – also known as the Bluetooth Address - and the length of the packet.

Next, I'll call our handy dandy advertising packet parser function which will figure out if the device has a name or is advertising a service UUID. I pass it the raw advertising packet and the length of the packet.

If it has a name, I'll print it out. If it is advertising that it has an LED Service, I'll print out that I found a device that I can connect to and I'll start the connection process with the ConnectDevice API and then I'll stop scanning.

The next interesting event occurs when a connection is made. That event is called CY\_BLE\_EVT\_GATT\_CONNECT\_IND. When that event happens, I'll turn on LED9 and start the service discovery process. Remember earlier I told you that I need to find the handles of the characteristics… that is exactly what happens inside of the StartDiscovery function.

Once that process is complete it will give you the event DISCOVERY\_COMPLETE.

The other interesting events, ERROR\_RSP and WRITE\_RSP occur when a write is successful or when one fails.

Now I need the BLE Task and Main. The BLE Task starts the stack and then prints out a message. In the loop, it calls process events. Then if there is data in the UART RX FIFO - in other words, someone has pressed a key - it will read the character. If the character was a plus it will increase the brightness and if it's a minus it will decrease the brightness.

The last function is Main… it just turns on the UART, starts the bleTask and starts the RTOS scheduler.

Alright, that was a bunch. Let's program this bad boy and see if it works.

You can see that I have two development kits here: the one that I just programmed; plus the one that is programmed with the Simple BLE Peripheral LED dimmer firmware.

Let's open a terminal window and see what it says. First, we see that it starts, then we see a bunch of BLE devices. Finally, it finds one we are looking for and it makes the connection. Then I can see the service discovery has completed and you can see that LED9 is active on the central and the Red LED has stopped blinking on the peripheral, so they're both connected… that's great.

Now let's see here… plus, plus, plus, plus… I see that the brightness is increasing… and then look, minus, minus, minus, minus… what do you know, it goes down.

If I hold the reset on the peripheral you will see that the central disconnects and starts scanning again after about 10 seconds… that's because the Connection Supervision timeout in the GAP Settings under the Connection Parameters of the peripheral is set to the default of 10 seconds. Once that timeout happens, it turns off LED9 and starts searching again. When I let go of the reset, quicker than anything it finds the peripheral and reconnects and now we're rolling again.

Personally, I think this is pretty damn cool. Hopefully you can see how we might make a remote control for the robot, which is exactly what we're going to do in the next video.

As always, you can post your comments and your questions in our PSoC 6 community forum or you can email me at alan\_hawse@cypress.com or tweet me @askioexpert with your comments, suggestions, criticisms and questions. And the first person who makes it to here who sends me an email – I'll send you a PSoC 6 development kit! The first person. Thank you very much. Goodbye.