3-3A: BLE Simple Central – Control of 3-2a LED Service

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 lesson we used your cell phone to talk to those devices. In this lesson I am going to show you how to create the other side of the connection. The "other side" is more properly called the central… or the GAP central.

The way that a BLE central works is that 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 then discover all of the services on that device… after that is done it can read and write the characteristics on the remote device.

You might guess from our previous lessons that there will be events that are called back to you to indicate that each of these steps has happened…. And you would be exactly right.

With this project I am going to show you how to take each of the steps I described earlier, and I am 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 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… or you can just follow along to learn.

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

Let's get started. To make things easier we are first going to open the simple BLE peripheral project and save the LED service so that we can use it again in this project. 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 on your desktop).

Now close that project and then create a new project called SimpleBLECentral. First let's edit the schematic. Add the BLE Component, a UART, and a digital output pin which we will call LED9. I like to use LED9 to indicate the presence or absence of a BLE connection. So, go ahead and change its name to LED9, make the initial state high – so that it starts 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 will 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 we are 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 you will right click on the Immediate Alert and pick delete. Now we have a blank client profile. The next step is to load in the LED service that we saved earlier. To do this right click on 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, then change the scan settings to always do fast scan… I ain't got time to wait!!! Now apply that.

I almost forget to assign the pins… go to the DWR and assign the UART to p5[0] and p5[1] and LED9 to P13[7].

As I have in all our projects I am going to use FreeRTOS and the Retarget I/O. 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 to tell it to include the project.h definitions and use UART_1_HW as the standard in/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 is to fix up main_cm0p.c so that it runs the BLE controller… so turn on the BLE stack… then in the infinite loop process events like so…

Finally… we are 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… and 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 first.

Let's see… returns a void… function name writeLED … takes a unit8_t as a parameter… which we will call brightness.

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

So now we know we are connected… so 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 … wow that is a mouthful.

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… wow!!! that is 16 bytes… that is a bunch. Rather than send 16 bytes in every transaction BLE creates a 1 byte alias for the UUID that is called a handle. The mapping of handles to UUIDs is sorted out during service discovery… which I will tell you more about a little later.

The bottom line is that for us to write the LED brightness characteristic we need to specify the 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 save in our structure.

Next, we need to give a pointer to the value… we need to tell it how long it is… 1 byte… those both go into our structure next. Then we need to tell it which connection. Remember when we set things up we configured only one connection at a time… so I hardcoded the connection. 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 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 is pretty simple. When you get an advertising packet you know the 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. This means you can scan through the packet and look at each of the fields to find what you are 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 project… first, open the project… then open the schematic… then double click on the BLE… then go to the GAP settings… and finally 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 bytes long… and the last is 18 bytes. On this tool you can see the other types of fields… All of these field types are specified by the Bluetooth SIG… but I am 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 what we the advertising packet looks like, let's go write the parser. First, I am going to store the name and name length… and UUID and UUID length into a structure… so let's declare that structure.

Then I am going to make the 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 I can now look for names and service UUIDs.

All right… the next function is the event handler. The function looks just like the other event handlers we 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 cause the event CY_BLE_EVT_GAPC_SCAN_PROGRESS_RESULT to occur telling us that it heard another device.

So… let's deal with that event…. On the UART I will printout that I heard a device … then I'll print out the BD 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 data and the length of the packet.

If it has a name… then I'll print it out.

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

The next interesting even 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 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 is 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.

Wow... OK let's program this bad boy and see if he 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 the one we are looking for and it makes the connection…. Then I 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 are both connected … that's great.

Now when I press ++++ I see that the brightness increases… and then ---- and it decreases.

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

I think that this is pretty damn cool… hopefully you can see how we might make a remote control for the robot… which is exactly what we are going to do in the next video.

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.