PSoC Academy: How to Create a PSoC BLE Android App
Lesson 4: Write the Firmware

Hello. I'm Alan Hawse. Let's get right into it. At this point we've created the schematic we've configured all the components, we've set the pins and now we're ready to work with the firmware. I'm just going to go slowly through the firmware. I'll explain what all the pieces and parts do, and then you'll be able to copy it for yourself, because we're going to post it onto Cypress.com.

So, first of all, I'll create a uint16 that will represent the finger position on the slider. This will be a global variable, so it's accessible inside of the BLE firmware. The second variable I create is called the capsenseNotify. This will be set to 0 when notifications are off, and it will be set to 1 when notifications are on.

The next thing I'll do is I'll create a function called updateLed. This function, when it's called, will update the GATT database with the current state of the LED. The first thing it does is if the BLE is connected – and you can find that out with the CyBle_GetState function – if it's NOT connected, then you don't want to update the GATT database, so you'll just immediately return from this function. If it IS connected, then you want to update the CYBLE_LEDCAPSENSE_LED_CHAR_HANDLE.

Now, remember, when you set up the profile and the services and the

characteristics inside of the GATT database, all inside of the BLE component, we named the LED characteristic led. When you do that, Creator will automatically build a #define that will reference you into that table – CYBLE_LEDCAPSENSE_LED_CHAR_HANDLE. Once you do that, you read the value of the pin with the read function and you flip it. The reason you flip it is the LED is active low, which means it's 0 when the light's on and it's 1 when the light's off. But that's not very sensible and everybody understands that 1 means on and 0 means off. So, on the user side it makes way more sense to send a 1 to turn it on and a 0 to turn it off. So, I'll do the handy dandy bang to flip the state.

The next thing you do is tell it that it's one byte, and then finally you actually write it into the GATT database with the Cyble_GattsWriteAttributeValue function.

So now we have a handy dandy helper function called updateLed. This function can be called anywhere in your source code, and all it does is, is if it is connected via BLE it reads the stage of the LED and writes it into the GATT database.

The next function that we will need is exactly same thing except for CapSense.

So, we'll start by checking to find out if we're connected. If we're not connected, then we'll just return out of this thing, because there's no need to update the database when you're not connected.

If you are connected, it will fall past the if statement and you'll just want to update the database. So, in the above function, we did CYBLE_LEDCAPSENSE_LED_CHAR_HANDLE. This time we'll want to talk to the CapSense handle. Happily, Creator created a #define for you, just like it did in the above case. This is based on the name you give it when you configured it in the characteristic when you're setting up the component.

The finger position was the global variable that we defined at the top of the program. This is a 16-bit unsigned integer, so first thing you need to do is you need to cast it into an unsigned 8-bit integer pointer, then you'll need to tell it it's two bytes, and then finally you copy it into the database using the GattsWriteAttributeValue function, exactly like you did above.

The difference in this function and the previous function is, do you remember when we set up the component, specifically when we set up the CapSense part of the component, we clicked on the notify button. Notify is a feature of BLE where the other side, the phone side, can say, every time you get a change in a

value, I'd like to know about that changed value. In other words, you can register for notifications of changes in the value.

I have a global variable called CapSenseNotify which gets set when the Client Characteristic Configuration Descriptor gets set If it's set, and the CapSense has changed, then I'll also, in addition to writing it into the GATT database, I'll send out a notification.

Cool, At this point I've created two helper functions, one to update the GATT database with the LED value, one to update the GATT database with the CapSense value. Now on to the next section.

When you start up BLE inside of Creator, which I'll show you in the main loop, first thing you have to do is you have to tell it, what is the event handler. So we're going to create a BLE event handler. I'll call this handler BleCallBack. So, the BleCallBack is essentially a giant case statement which allows me to deal with the myriad of different events that get called from the BLE firmware.

The first two events that I'll handle are the stack turning on, or the stack being disconnected. In other words, when you first start going, it gives you the event CYBLE_EVT_STACK_ON. And then after you've been connected and you get

disconnected, that event is called CYBLE_EVT_GAP_DEVICE_DISCONNECTED.

I'll handle these two events exactly the same way. What I'll do is I'll turn off notifications and then I'll start the advertising again I will also start the PWM that drives the blinking blue LED so that I get the blinking light telling me that I'm advertising. There are two situations where you want the light to blink. You want it to start blinking when the chip first turns on to indicate that it's disconnected. The other time you want to start the blinking is after you've been disconnected by the remote side, specifically, the Android app side of this equation. So when you've gotten disconnected, the BLE stack will send you the CYBLE_EVT_GAP_DEVICE_DISCONNECTED message. And when you get that message, you start the PWM going so the blue light blinks again.

The next event that we'll need to handle is, what happens when you've gotten a connection established from the remote side? Well, the first thing that you need to do is you need to update the GATT database with the current state of the LED so that the Android app can see whether the LED is currently on or off. So, it's convenient, we wrote a little function, that grabs the state of the LED and puts it into the GATT database. So we'll just call that function.

The other thing that we need to do is update the GATT database with the

CapSense value. This is so the remote end can read the state of the LED and it can read the state of the CapSense. That's handled automatically for you by our BLE stack firmware which is completely free and comes with the BLE component.

The other thing we want to do is, after you've gotten a connection established, you'll want to turn off the blinking blue light. So, the light will blink blue when you're disconnected, and it will be turned off when you're connected.

The next event that we can deal with is the Write event. This is called CYBLE_EVT_GATTS_WRITE_REQ. This event gets called when the remote side wants to write into your GATT database. When it asks to write into your GATT database, our stack will tell you which characteristic it's trying to write, and what you need to do is look at all of the characteristics and do something slightly different based on which characteristic the other side is trying to write into.

So, in the first case you say: Is the CYBLE_LEDCAPSENSE_LED_CHAR trying to be written? In other words, did the remote side try to change the state of the LED? If the answer to that is yes, then, go ahead and update your database with whatever the remote side wrote into you by calling the

GattsWriteAttributeValue, just like you did in the UpdateLED function. The other thing you want to do is after they've written into your LED characteristic, you want to go ahead and to turn on the LED or turn off the LED, based on whether they wrote a 0 or a 1.

Remember I told you the red LED is active low. So when they write a 1, you want it to turn on, and when they write a 0, you want it to turn off – except for the fact that when you write a 0 it'll turn on because its active low. So you'll need to use the bang to flip the state of what they wrote. The bang for those of you who aren't big programmers out there is just the exclamation point.

Okay. So we're talking about the different Write requests. The first Write request is if they try to write the LED characteristic. The second Write request is if they try to change the notification flag for the CapSense characteristic. That's called CYBLE_LEDCAPSENSE_CAPSENSE_CAPSENSECCCD_DESC_HANDLE. Remember in the previous case we changed the name of the characteristic CapSense descriptor to be capsensecccd so we wouldn't have to type as much. When that happens, you want to go ahead and write into the GATT database with the WriteAttributeValue function, and then you want to turn on the global variable flag, or turn off the global variable flag for notifications, based on

whether they turned it on or off.

This way, when you call the updateCapsense function other places in your firmware, if they've said they're interested in notifications, the notifications will be sent out. So we'll modify the global variable that we're using to keep track of the CapSense Notify being turned on or off.

The last thing you need to do, the BLE spec says that when you get written into, you need to give a response. So, in either one of the cases, whether they write into the LED or whether they write into the CapSenseCCCD, you need to send a response. To do that you use the CyBle_GattsWriteRsp function.

At this point we've got the update LED function, we've got the update CapSense function, and we've got the BLE stack event handler function. The last function that you need is the main function, and it has two basic parts. It has initialization and it has the main loop. In the initialization, you'll need to turn on the global interrupts. This is done with the CyGlobalIntEnable macro. All of these things that we're doing, both BLE and CapSense, are interrupt-based, so you'll need to have the interrupts on.

The next thing you'll need to do is start the CapSense component, and then

you'll need to get the CapSense component to initialize its baselines. Initializing the baselines is a function that gets rid of the baseline noise that's in your system as well as dealing with the parasitic capacitance attached to the sensors, in this case the slider that's attached to your pins. That's the first step of getting the CapSense going.

Then, the main loop is very trivial. If the CapSense is not busy – in other words, if the CapSense has completed and it's not scanning anymore - then grab the finger position using the GetCentroidPos function, update your baselines, start the scan again, and then call your helper function that we talked about earlier. That function will take the finger position and it write it into the characteristic for CapSense if you're connected. If it's not connected, then it doesn't do anything. So, we've dealt with the CapSense, and you'll need to do that every time through your infinite loop.

The next thing you'll need to do is you'll need to give BLE the opportunity to process its events. That's done with the CyBle_ProcessEvents function – very easy. The last thing you want to do is tell the BLE that it's welcome to go to sleep if it can, in which case you'll save power. You do that with the CyBle_EnterLPM function.

That's it. That's your whole loop. So, from the top, we have a function called updateLed which will update the GATT database with the state of the LED. We have a function called updateCapsense which will update the GATT database with the state of the CapSense. We have a BLE event handler that will handle the stack turning on or a disconnection, and it will handle writes of the characteristics from the Android phone application. And the last thing is we have the main loop, where we initialize the CapSense, run the CapSense in the infinite loop and process the BLE events. That's pretty simple.

In the next video, I'll show you how to debug your firmware and make sure that it's working properly using CySmart on your Android phone as well as CySmart on the PC.