PSoC Academy: How to Create a PSoC BLE iOS App
Lesson 7: Deep Dive into App

00:00:08 Welcome back. I'm Alan Hawse. This is Cypress Academy. Now we're going to dig through the iOS app, and I'm going to show you the three pieces. The first piece is the storyboard. The storyboard is the view in the MVC. It controls the buttons that trigger each of the actions. It controls the two switches. One to turn on the LED, and one to turn on the notifications. And it contains the label that

00:00:36 represents the CapSense value. You start by copying the buttons onto your canvas. Or the labels onto your canvas. Or the switches onto your canvas. I've already done all of that in this example. There's a switch. And you can just drag it on your screen. The first button is entitled Start Bluetooth. And then Search for Device. And then Connect to Device. And then Discover Services. You can see

00:01:08 that the first one is blue. That means it's enabled. And you enable it through the Property Inspector. User Interaction is enabled. So these buttons are not enabled. They're gray, which means if you touch them they don't do anything. And as you recall from the demo, the first one will be blue. It'll be active. And then after the actions are done, I'll turn it off and I'll turn on the other one. And I

00:01:39 do that with software which I'm going to show you in the next step. So now I'll go to the Assistant's Editor. And you can see that each of the buttons are connected into the view controller. So the Start Bluetooth button is connected to this IBOutlet called

startBluetoothButton.

00:01:58 The Search for Device is connected to the searchButton. The Connect to Device is connected to the connectDeviceButton, et cetera. And you can see that each of the buttons have an action. The startBluetoothDevice has the startBluetoothAction. So that's how the connection is made between the view and the view controller. Now I'm going to show you some more detail on

00:02:28 the view controller. The first thing I've done is define a global structure which I call RCNotifications. This structure just has a number of members: Bluetooth ready; Found device; Connection complete. These are just strings that I'll use to send notifications with. They're in reverse Internet order, org.elkhorn-creek.simplecapsense.bluetoothReady. This is so you don't accidently collide with another notification that's going on in your

00:03:00 system. Elkhorn Creek is actually in my backyard in Kentucky, and so I do a lot of my personal projects with Elkhorn-Creek.org. Inside of the view controller, the very first thing that I do is, when the view loads up for the first time, I tell the NSNotificationCenter, which is the Apple-provided iOS app that lets you listen for notifications from other models in your system, that I want to

00:03:30 listen for the RCNotifications.BluetoothReady message. And when that message happens, I want to call self.BluetoothOn. So

essentially what happens is when the model sends out the
00:03:47 message, RCNotifications.BluetoothReady, I'll call the self.BluetoothOn. Self.BluetoothOn, what that function does is turns on the next button in the sequence. So each of these buttons will send a message back to the model. The model will talk to the CBCentralManager, which will do its thing. Then the model will be notified by the CBCentralManager through the delegation system that something has happened. When that happens, it'll
00:04:23 send an NSNotification that the view controller is listening to. So let's go slowly through these. The first one as you'll recall is the startBluetoothAction. That's the first button at the top of the screen. When that happens, what you want is you want the model which we're going to look at, to turn on the CentralManager. So let's flip to the model. So I'll click on the Bluetooth Neighborhood which is the class I created to represent the peripheral and to
00:04:56 represent all of the Bluetooth world. And when the function is called, startupCentralManager from the view controller. All it does is, it says 'give me a new CBCentralManager object'. That object represents the CentralManager that's inside of the iOS, and then I'll store that in a local variable which I'm calling centralManager, which is of type CBCentralManager. So what happens is, it tells iOS go get it going. So the hardware will turn on. It'll start its

thing, and then once it's going, it'll send the message back to the
00:05:38 model called centralManagerDidUpdateState. So when that function is called by the CBCentralManager, I'll look at the different states that it gives me back. It can give me PoweredOff, it can give me Unknown, it can give me Unsupported, it can give me Resetting. All of those things are bad. The state that we're looking for is Powered On. Once the CBCentralManager is powered on, I send the NSNotificationCenter a message called
00:06:09 RCNotifications.BluetoothReady. What that does is, it says anybody who knows about this model, you should be aware that I've changed. And what's changed is the CBCentralManager is on. And it's ready to have commands sent to it. Back in the view controller, when that happens you registered that you want to call the function self.bluetoothOn. So that calls self.bluetoothOn,
00:06:37 which enables the Search button which is the next button down on the screen. Then when the user presses the Search button, it calls the action searchAction which tells the model – the first thing it does is it tells the model I want to discover the device. So back in the model, you can see the discoverDevice function, and what it does is, it says CentralManager, start scanning for peripherals with
00:07:07 services, and the service that I'm interested in is the BLEParameters.capsenseLedService. Now you remember, when we

set up the BLE component, we created the custom service. And it

00:07:21 had the UUID which I call blah, blah, blah, blah, blah F0. Well, I copied that blah, blah, blah, blah F0 into another global structure which I call BLEParameters, and the first of the global structure parameters is the capsenseLedService which is a CBUUID - Core Bluetooth Universally Unique Identifier - that matches the hex code blah, blah, blah, blah, blah F0. And essentially, what this does is, it tells the CBCentralManager there's a lot of Bluetooth devices out there,

00:07:59 and there may very well be a lot of Bluetooth devices out there. What it says is I'm not interested in talking to those devices. I'm only interested in a device that is advertising that it supports this service. So what happens then is the CBCentralManager starts listening. And as soon as it hears a device like that, it calls me back. And it calls the function CentralManager didDiscoverPeripheral. And that calls me back with an object of

00:08:32 CBPeripheral type. Now the CBPeripheral is an object that represents a peripheral out in the world. And it doesn't know much about that peripheral other than what it heard in the advertising packet. So what I do is, if I'm not already connected, then I say that I found a new peripheral that's advertising those services. I save in a local variable that peripheral. So now I have a local

reference to a peripheral that's advertising the correct services.

00:09:05 And then I post a notification. I send out a notification to the view controller that I've now found a working device, and then I tell the CBCentralManager stop scanning. Not interested in talking to anybody else. So what you could do, is if you were talking to more devices, you could have an array of these peripherals. I'm not doing that here. The first one I hear, that's the only one I'm going to talk to. Okay, inside of the storyboard when the user presses

00:09:37 Connect to Device, it triggers an action inside of the view controller called the connectDeviceButton. So it actually calls the connectDeviceAction which turns off the Connect button, and tells the BLE object, our model, to actually make the connection to the device. Inside of our model, this says the connectToDevice function tells the Central Manager, go ahead and make a connection to that peripheral. Once the back and forth of the BLE

00:10:10 radio protocol is complete, the Central Manager calls you back with, yep, I did connect to the peripheral. Once that happens, you can notify the view controller that you now have a complete connection. And I do that with the post notification of RCNotifications.ConnectionComplete. Inside of the view controller, when it hears that message, ConnectionComplete, it calls the connectionComplete method. The connectionComplete method

turns on the services Discovery button. And turns on the

00:10:49 Disconnect button. Then once the user presses the Discover Services button, we go through another round. It calls the discoverServicesAction and disables the Discover Services button. Back in the model, in the Bluetooth neighborhood, when you call discover Services, we've switched from calling CBCentral methods to calling CBPeripheral methods. Once you have a connection established to make your communication to the specific peripheral happen, you don't call the central to make those happen.

00:11:22 You call the peripheral to make those happen. So the first thing you do is you tell the peripheral I'm interested in what services you've got to provide. That tells iOS BLE to go do the service discovery process. The BLE service discovery process talks with the BLE stack inside of the Cypress PSoC 4 BLE. And they work out the back and forth communication to figure out what are the services that are available for use.

00:11:52 When that process is complete, it calls this function peripheral didDiscoverServices, and what it does is it adds a list of CB services and it attaches those CB services to your peripheral object. You can then iterate over all of the services and you can look at each individual service. In our case, the service that we're looking for is the capsenseLed Service. Which I

00:11:25 put into that global structure that I call BLEParameters, and I say if the service UUID matches the capsenseLed Service, then I want to save that service because I know that's the service that's going

00:12:38 to have the two characteristics. I declared the capsenseLed Service in the global structure BLE parameters, and as you'll recall, I called it blah, blah, blah, blah, blah F0. So here it is. That's the capsenseLed Service. Once that's complete, the Discover Characteristics button is now turned blue because the user interaction is available. So the user, you or me or whoever, can press the Discover Characteristics button. When you press that

00:13:10 button, it calls this discoverCharacteristicsAction, and what that does is it tells the model I'm interested in all of the characteristics that are associated with the capsense service. So back in the model again, in the Bluetooth neighborhood, you can say that I say have this function discoverCharacteristics which gets called from the view controller, and that function calls the CB peripheral method discoverCharacteristics forService, And we're

00:13:43 specifically interested in the characteristics that are associated with the capsenseLed Service. That triggers the peripheral. It talks back and forth between the iOS and the peripheral. When that talking back and forth with BLE is complete, it calls this function peripheral didDiscoverCharacteristicsForService. And what will

happen is, is your CB service will now have an array of CB characteristics associated with it. You can then scan through that array of characteristics and look at each characteristic. If the
00:14:21 characteristic's UUID matches the UUID of the characteristic you're interested in, you can save it. And in our case, we're interested in the two characteristics: the capsense characteristic and the LED characteristic. And in my global table, I said the LED characteristic UUID is blah, blah, blah, blah, blah F1, and the capsense characteristic UUID is blah, blah, blah, blah, bah F2. Once I've scanned through all of the characteristics I found, I save
00:14:55 a reference to the characteristic if it's the capsense characteristic, and I save a reference to the LED characteristic if it's there. Once you've got all this back and forth done, you've got the BLE turned on – the Central Manager turned on. You found a device. You connected to the device. You found all the services on the device. You found the service, the specific service that matches the capsenseLed Service. You discovered the characteristics that are
00:15:26 associated with the capsenseLed Service. Specifically the LED service and the capsense service. You now have a complete connection and you can talk to individual characteristics. So I send out the notification - all of my characteristics scan is done. I have a complete connection. Now if you remember, on the

storyboard you've got two characteristics. You've got the red LED which has got a switch associated with it and you've got the capsense which has a label which starts as questions marks, and

00:16:00 notify. So let's start with the red LED. When the red LED switch is switched, it calls this function ledSwitchAction. When the switch action is called, I ask the switch is it on or is it off? If it's on, then I write the LED characteristic with one. And if it's off, I write the LED characteristic with zero. Now notice that one and zero, those are magic numbers which if you're doing a project for me, I'll slap your hand. But we're expedient in this case. These writeLedCharacteristic,

00:16:38 they're just functions that are part of the model. And inside of the model I do a BLE command when I get that call. And that BLE command is called writeValue. So I take either the one or the zero, I turn it into bytes with this NSData function, and then I write those bytes into the ledCharacteristic. And I ask for a write with response. You remember, when we configured the BLE characteristic inside of the Creator component, I said it was a read characteristic and a write characteristic. Write always means write

00:17:19 with response. So I write with a response. That's it. The iOS CB peripheral and all the magic that's inside of the cell phone, and all of the magic that is PSoC 4 BLE talk back and forth, and then they

trigger down in the firmware that write. So that's pretty cool. Now the other thing we've got going on is the CapSense value. Now we're interested in CapSense, a read from the device, essentially.

00:17:51 And specifically the read that we're interested in is the notification. So for us to get notifications back from our board, we need to tell our board that we're interested in notifications. So when this switch is switched, it calls an action in our view controller called the capsenseNotifyAction, and all the capsenseNotifyAction does is it calls the model and it says please write the CapSense notify, and tell it whatever the sender on is. Now that switch when it's on, it's

00:18:27 true and when it's off, it's false. So when it's on, we want notifications and when it's off, we want no notifications. So when we look down inside of our model, we see that the writeCapsenseNotify function that we called from our view controller takes in a boolean, either the on or off and then sends out a notification. setNotifyValue is a member function of the peripheral class, and what it does is communicates with the peripheral that says please turn on

00:19:02 notifications so that I'll be notified when the CapSense changes. What happens then is, the capsense LED board that we built will start sending out changes. The iPhone hears those changes because it's connected, and when it hears the change, it calls this

function Peripheral didUpdateValueForCharacteristic. So when that happens, you look and see which characteristic changed. In this case, we were interested in the capsenseCharacteristic. So we got a note from the iOS that said a value inside the characteristic changed. You say which characteristic was it? It matches the

00:19:42 capsenseCharacteristic. Then you have to unwind bytes and so you do this little iOS rigmarole where you turn bytes that come in NSdata into an integer, then you save the value, and then you send out a notification to your view controller: the CapSense value has changed. Then inside of the view controller, I told you that we registered a notification called addObserverForName, RCNotifications.UpdatedCapsense, and when you get this

00:20:15 notification, you want to update the CapSense value on the screen. And all that you do down in your view controller is you ask your model what's the value, you turn it into text, and then you put it on the screen. That simple. So your board sends out the notification. It works its way through the Cypress BLE stack out over the radio into the iPhone through the iPhone BLE stack. Then it calls your app back. Says this characteristic changed. Then your

00:20:48 characteristic sends a notification. The view controller reads the value that it saved locally. Turns it into text and puts it on the screen. That's it. So once again, the model is the

CBCentralManager, as well as the peripheral that represents your board. The view controller takes button clicks and tells the model to do things. And when it gets notifications back from the board, it makes the display. That's all there is to it. I'll post this project, and I'll post the firmware, and you're welcome to send me questions at Alan_Hawse@Cypress.com.