All right, now we're ready to walk through the schematic. I'll show you the quadrature encoders that drive the H-Bridge, the PWMs, et cetera – all the parts on the schematic. Then I'll show you the configuration of the BLE, and then I'll slowly walk through the firmware. Each of the motors have got a PWM that drive them. There is a left one and a right one. They're configured exactly the same. One drives the pin for the left motor and one drives the pin for the right motor.

The PWM is simple. It's just a period of 100 and a compare value of 50. I run the duty cycle between 0 and 100 to match 0 and the 100 on the phone. The quadrature encoder is used to decode the quadrature outputs of the motor. Quadrature is a train of pulses that are 90 degrees out of phase. With a quadrature encoder you can tell the speed something is moving as well as the direction it's moving.

The TCPWM blocks inside of the PSoC can decode quadrature. You just give it the left pulse and the right pulse. So each motor has two trains of pulses coming out of it. I call those qd1a and qd1b for the left motor, and qd2a and qd2b for the right motor. When you look at the quadrature encoder you just configure it as a TCPWM. I put it in 1X mode which means I count one pulse and not the edges where you could get 2X or both edges where you could get

4X. That's all there is to it. The two inputs are level sensitive.

The next thing that I do is control the pins of the input to the H-Bridge. The H-Bridge needs to have power and ground. It doesn't use very much current to control the chip that is the H-Bridge, and the PSoC is perfectly capable of creating enough current so I power it with the PSoC, and I use the ground of the PSoC to be the ground of the H-Bridge. And I do the same thing with the right one.

This enables me to have those nice nifty connectors that plug directly into the port with all six wires on it. The other signal is a software-controlled direction pin. The left and right H-Bridge have a pin that when you flip the pin one way it runs the current through the H-Bridge one way. And when you flip it the other way the current runs the other way through the H-Bridge. This allows you to control the direction of the motor via software.

I decided I needed an emergency stop for this crazy robot so I connected the hard switch that's on the board to an interrupt. When I press the pin it immediately stops the robot. This is for when my son is running around going crazy and the robot is going back and forth like who knows what, so I click the button and it immediately stops the robot.

The last two things I put in the schematic are a PWM to blink the LED so when you're not connected I blink the LED red and as soon as you connect I turn off the LED. Finally, I put in quadrature encoders count 64 pulses per revolution, and I need to turn these pulses into revolutions per minute. So every 187 milliseconds, I trigger an interrupt to go find out how many pulses have occurred in the quadrature encoder in the last 187ms. I then multiply that number by a scaling number required to get the tachometer for the left and for the right motor.

Now I'm going to show you the custom service that I created to drive this thing. Just as before, we have it setup as a custom profile, a GATT server, as well as a GAP peripheral. The cell phone Android app, which I'm going to show you next, controls the device. It's the GAP central. The PSoC 4 BLE is the GAP peripheral.

I created a custom profile which I call the MotorService. The MotorService has four characteristics; the left speed, the right speed. Those are signed integers that are eight bits long. Here I'll show you one. See, it's a signed integer, eight bits. A signed eight bit integer can do any number between minus 128 and plus 127. That's more than enough to show the PWM of minus 100 to plus

100. This characteristic is writable and readable. So from the app when I want to spin the motor in the positive direction, I write a positive number up to 100; 100 means 100 percent duty cycle, 50 means 50 percent duty cycle, and obviously zero is off.

When I put a negative number in I use software to flip the pin that does the motor control direction so that the current goes the other way and the motor runs in the reverse direction. I then take the absolute value of that characteristic and I write that value into the compare of the PWM. So the left speed and the right speed are configured exactly the same way.

Notice I put in the Characteristic User Description so you can use a GATT database query to find a human-readable form of the user characteristic.

Now I create the left tachometer and the right tachometer. They both have CCCDs so you can ask to be notified any time the tachometer changes. These are setup as 32 bit signed integers - four bytes long. You can read them and you can ask for notify. So inside of the firmware I calculate the number of RPMs that the motor is going either in the positive direction or in the negative direction. I store it in the GATT database in the characteristic and it's setup for notify. So in your Android cell phone app you can say I want to be notified

when the tach changes. And then every 187 milliseconds in the interrupt that I do the calculation, I update the GATT database and broadcast the notification.

You'll notice that I did the same thing as I did earlier with the UUIDs. I left the Creator default which was blah-blah-blah-blah, F something, and I changed the service to F0, the left characteristic to F1, the right characteristic to F2, the left tachometer to F3, the right tach to F4. So I take those four UUIDs and I can put them into my Android app.

In the GAP settings I gave the thing a name and in this case I call it Robot. I advertise for forever. I've got a whole bunch of batteries so it's really not that big a deal.

And inside of the advertising packet I advertise the name of the robot and I advertise the fact that I have a motor service in there, and the motor service is my own custom UUID. This is so on the Android phone I can only listen to BLE devices that are advertising that they have the motor service UUID. This simplifies the connection process.

All right, now I'll walk through the firmware.

I have it in one big long file, and I've got several flags. I've got the left and right tachometer notify flags. I've got the current value of the left and right tachometer which I save as global variables. I also keep the current left and right speed. I have an interrupt service routine which gets triggered every time the user presses the button on the board. Remember, that's the emergency stop. All that does is turn off the motors by setting the motor flag which I can read later on.

I have a function called updateSpeed just like I had a function called updateLed and a function called updateCapsense in the previous project. All that does is if there's a connection then it takes the current left speed and stores it in the GATT database under the correct characteristic. It takes the right speed and stores it in the correct characteristic. This is because you could walk up on a robot that's already moving but wasn't currently attached and you could read the values that are currently driving the PWMs.

The next function I have is setSpeed. It takes this enumeration, which I setup earlier, that's either left or right. It's called by the BLE Stack Event Handler when there is a write to either the left characteristic or the right characteristic; the left speed or the right speed that is. It decides what the absolute value of the speed is. If it's negative then it makes the motor spin backwards, and if it's

positive it makes the motor spin forwards. It does that by setting the pin on the H-Bridge that steers current either the forward way through the motor or the reverse way through the motor and that's simply a logic 0 or a logic 1. If somebody tries to set the speed above the maximum value of the PWM, which is 100, then I just say no way and I ignore it.

Then what I do is I switch to find out if we're talking about the left motor or the right motor. I start by setting the pin to the correct direction. I set the PWM compare value to change the duty cycle of the PWM, either the left one or the right one, then I save the current speed in either the left or the right speed variable. I then update the speed in the GATT database with the updateSpeed function.

The next function I have is an ISR which triggers every 187 milliseconds. That ISR says the next time you get a chance in the main loop you should read the quadrature decoders and find out how fast you're going. And that's all this function is – a trigger flag which I read in the main loop telling me, okay, go query the quadrature decoders.

Now I have another function which behaves just like the updateLed, which behaves just like the updateCapsense, which behaves just like the

updateSpeed, which updates the left and right tachometers. So if there is a connection then it follows through. If there is not a connection then it stops. Then what it does is it sets the left tachometer characteristic to be equal to the current tachometer and writes into the database. Then if notifications are turned on, it broadcasts the notification for the tachometer. Then it does the exactly same thing for the right one. It updates the GATT database, then if the right notify is on then it sends out the notification for the right one.

The next function is an inline function, which means the compiler puts it right there in your code where you call it. The first thing I do is turn off the flag. I read the counter in the left tachometer. I read the counter in the right tachometer. I multiply it by the scaling factor. And then I save that into my global variable which is keeping track of the speed of the left motor and the speed of the right motor, and then I put the quadrature encoder back to its middle value. So when the motor is going forward it counts up from the middle value, and when the motor is going backwards it counts down from the middle value. So every time I read it I put it back to the middle so I know whether it's going forward or backwards.

The next function is just exactly like I showed you in the previous example. When the stack turns on I turn the left and right tachometer's notifications off.

I start the advertising and start blinking the LED that says I'm not connected. Once I get a connection event, I update the RPMs, I update the speed, and I stop the blinking.

When I get a write request I need to decide whether they're trying to write the speed into the left, the speed into the right, or whether they're trying to turn on notifications in the CCCDs. So I look at it and if they write into the left speed I use my setSpeed function with the left and I save whatever they gave us and if they are talking about the right characteristic then I do the same thing. I just set the speed with my helper function for the right PWM.

Then if they're trying to write into the CCCD I update the GATT database with that value and I store it in the notification. So if they set it to one then I turn on notify and if they set it to zero I turn off notify. Then like all writes you have to respond so I send out the response.

When you press the button to turn off the motors I have a little thing that if the motors are running then I stop them. I just turn off the PWMs. If the motors are not running then I turn on the PWMs so I toggle between them. When you press the switch it turns them on, if you press the switch again it turns them off.

Then the last thing is the main loop. The main loop starts up the switch interrupt handler. It starts the two PWMs and sets their speed to zero so both motors start in the stopped state. In other words, the system turns on with both PWMs set to zero percent duty cycle so that the motors are not turning. I then enable the quadrature encoders and tell them to get going. I start the tachometer. I start the LED to flashing. I turn on the BLE with the CyBle_Start, and I tell it the callback we just talked about, and then in in my main loop I handle changes in the motor. I process the tachometers. I run the BLE process. And I try to put the BLE to sleep to save power. That happens infinitely.

That's it for the firmware. It looks just like the firmware you've already seen before and I don't think there is much in the way of surprises. But as always you're welcome to email me at alan_hawse@cypress.com and I'll tell you more about it. This project will also be posted on the internet so you can download it and look at it for yourself. In the next video I'm going to tell you more about the Android app and take you through its creation process.