00:00:08   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. 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 the left one and the right one. They're configured exactly the same. One drives the pin for the left motor. One drives the pin for the right motor. The PWM is simple.

00:00:38   It's just a period of 100 and a compare of 50. And I run the duty cycle between 0 and 100 to match 0 and 100 on the phone. The quadrature encoder is used to decode the quadrature outputs of the motor. Its 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 and the direction that something is moving.

00:01:09   The TCPWM blocks inside of the PSoC can perfectly well 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. So 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 the speed.

00:01:49   That's all there is to it. The two inputs are level sensitive. The

next thing that I do is I 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 for that so

I power it with the PSoC. And I use the ground of the PSoC to be

the ground of the H-Bridge. And the same with the right one.

00:02:22   This enables me to have those nice nifty connectors that plug

directly into a port with all six wires on it. The other signal is a

software-controlled direction pin. The left and the right H-Bridge

have a pin that when you flip the pin one way it runs current

through the H-Bridge one way. And when you flip it the other way

it runs the current the other way. This allows you to control the

direction of the motor via software. The other thing I put on the

system is I connected the pin that's connected to the switch on the

chip to an interrupt.

00:02:56   When I press the pin it immediately stops the thing. So this is

when my son is going crazy running around it's easy to stop the

thing. The last two things I put on this design in my schematic is

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. The other

thing is the quadrature encoders count 64 pulses per revolution.

00:03:25   And I need to turn those 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. And then I

multiply that number by the scaling number required to get the

tachometer for the left and the right. Now I'm going to show you

the custom service that I created to drive this thing. Just as

before, we have this setup as a custom profile, a GATT server, and

a peripheral.

00:04:00   The cell phone app, which I'm going to show you next, controls the

device. It's the 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. And here

I'll show you one; signed integer, eight bits.

00:04:29   A signed eight bit integer can do any number between minus 128

and plus 127. That's more than enough for 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 positive I write a positive

number up to 100; 100 means 100 percent duty cycle, 50 means

50 percent duty cycle, and obviously zero is zero percent duty

cycle.

00:05:03   When I put a negative number in I use software to flip the pin to

run the motor the other way and then I take the absolute value

and write that value into the PWM. So the speed left and the speed

right are configured exactly the same way. Notice I put in the

Characteristic User Description so you can query to find out a

human-readable form of the user characteristic, of the

characteristic.

00:05:30   Now I've created 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 positive or negative.

00:06:00   I store it in the GATT database in that characteristic and it's setup

for notify. So in your cell phone app you can say I want to be

notified when the tach changes. And then every 187 milliseconds

that the interrupt calls the calculation I update the GATT database

and send out 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.

00:06:30   And I changed the service to F0, the left characteristic to F1, the

right characteristic to F2, the left tachometer to F3, and the right

tachometer to F4. So I take those four UUIDs and I can put them

into my 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 not a big deal.

00:07:02 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 iPhone 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.

00:07:29 I have it in one big long file. 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. And then 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. And all it does is turn off the motors by setting this motor flag which I read later on.

00:08:04 I have a function called updateSpeed just like I had a function called updateLed and a function called updateCapsense. All that does is if there is 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 to and you could read the values that are the current settings of the PWMs.

00:08:40   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 right 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 that is negative then it makes the motor spin backwards. And if it's positive it makes the motor spin forwards.

00:09:11   It does that by setting the pin on the H-Bridge that steers the 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.

00:09:40   I set the PWM compare value to change the duty cycle to either the left one or the right one, and then I save the current speed in either the left or the right speed variable. And then I update the speed in the GATT database with the updateSpeed function. Now I have an ISR which triggers every 187 milliseconds. And 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.

00:10:12 And that's all this function does is it triggers a flag which I read in

the main loop that tells me, okay, it's time to go query the

quadrature encoders. 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.

00:10:42 Then what it does is it sets the left tachometer characteristic to be

equal to the current left tachometer and writes it in the database.

Then if notifications are turned on, then it broadcasts the

notification for the tachometer. Then it does the exact same thing

for the right one. It updates the GATT database, and then if the

right notify is on then it sends out the notification. The next

function is an inline function.

00:11:12 Inline means the compiler puts it right there in your code where

you call it. The first thing I do is I 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.

00:11:44   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 after 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 I start

blinking the LED that says I'm not connected.

00:12:16   Once I get a connection event I update the RPMs, I update the

speed, and I stop the blinking light. When I get a write request I

need to decide whether they're trying to write into the left speed,

the right speed, or whether they're trying to turn on the

notifications in the CCCDs. So I look at it and if they write into the

left speed I use my setSpeed function with left and I save whatever

they gave us. If they are talking about the right characteristic then

I do the same thing.

00:12:49   I just set the speed with my helper function for the right PWM.

Then if they're trying to write in 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 the notify and if they set it to zero

then I turn off the notify. Then like all writes you have to respond

and 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.

00:13:23   I just turn off the PWMs. If the motors are not running then I turn on the PWMs and I toggle between them. So the switch will turn off the motors if they're on or turn them on if they're off. That's nice and simple. Then the last thing is the main loop. The main loop starts up the switch interrupt handler. It starts the two PWMs and sets the speed to zero so both the motors start stopped.

00:13:51   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 is its callback, and then in in my main loop I handle changes in the motor.

00:14:22   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 seen before. 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 be posted on the internet

so you can download it and look at it for yourself.