Testing the bus-pirate npm module with Resin.io

When you’re writing code, unit tests can be a very nice, fuzzy security blanket– especially when paired with continuous integration, code coverage, and other monitoring tools.

But what do you do when you’re writing code that controls hardware? Unit testing can be a nightmare, even if you mock out the actual hardware peripheral!

But even if you get tests written, sometimes mocking out the hardware peripheral can cause more problems than it solves– if there’s a firmware update, and you’re only testing against a mock, it might be a while before you notice any issues!

I came across this problem when writing a library to control the Bus Pirate with Node.JS– the unit tests were passing, but the example code wasn’t working! Turned out my mock sent an ever-so-slightly different response code than the actual firmware.

Eventually, the idea occurs to write tests that run against the hardware itself. But that introduces a whole new set of issues:

- What if you want to test multiple hardware versions of one

peripheral, or test the hardware in different physical configurations?
- How do you make sure the hardware tests run when you push new code?
- How do you make the testing rig constantly available without carrying it around everywhere?

Luckily, Resin.io provides an eloquent platform to launch our testing code on–

- We can launch our tester on as many devices as we need without having to manually install it on each one– we can have different physical configurations on each one!
- Using the public URL from the resin dashboard, we can create a git hook to run our tester at the same time as any of our other continuous integration solutions
- Using resin, our testing rig(s) are always available as long as they have wifi access!

To do this; I used the following hardware:

- A BeagleBone Green Wireless (with built-in WiFi chip)
- A BusPirate v3.6 or v4.0, plugged into one of the BeagleBone's USB ports
- Either a TCS72435 RGB Color Sensor or an Arduino Uno set to echo UART traffic wired to the Bus Pirate

Let’s get started!

First, we’ll take an example test from the bus-pirate-tester test suite. bus-pirate-tester uses Mocha for a test suite and sinon.js for function spies.

```
describe('start()', () => {
  it('should fire the ready event when the bus pirate sends BBIO1', (done) => {
    let eventHandler = sinon.spy()

    busPirate.on('ready', () => {
      eventHandler()
    })

    busPirate.start()

    console.log(busPirate.inputQueue.buffer.toString())

    setTimeout(() => {
      assert(eventHandler.called, 'Ready event handler was not called')
      done()
    }, 1900)
  })
})
```

This test starts up the Bus-Pirate, and waits for a ‘BBIO1’ response, and checks that a `ready` event is fired when this is complete.

Using a Dockerfile to set up our device image

Resin allows you to use a Dockerfile template to customize your device image. This way, we can clone the latest version of the bus-pirate library, get the USB serial port started, and install the needed npm modules automatically before the initial run of the tests. Let’s take a look:

```
# base-image for node on any machine using a template variable,
# see more about dockerfile templates here:
http://docs.resin.io/deployment/docker-templates/
# and about resin base images here:
http://docs.resin.io/runtime/resin-base-images/
# Note the node:slim image doesn't have node-gyp
FROM resin/%%RESIN_MACHINE_NAME%%-node:slim

# install git for cloning the repo, python and build-essential
# for compiling node-serialport on the BeagleBone
RUN apt-get update && apt-get install git python build-essential

# Defines our working directory in container
WORKDIR /usr/src/app

# clone the node-bus-pirate repo and install its dependencies via
npm
RUN git clone https://github.com/nodebotanist/node-bus-pirate.git
./
RUN JOBS=MAX npm install --production --dev --unsafe-perm && npm
```

```
cache clean && rm -rf /tmp/*

# next we install the modules needed for our testing server
RUN JOBS=MAX npm install express simple-git && npm cache clean &&
rm -rf /tmp/*

# Enable systemd init system in container
ENV INITSYSTEM on

# These commands enable us to use our USB serial ports
RUN udevd --daemon && udevadm trigger

# Copy our working directory files over to the container
COPY . ./

# server.js will run when container starts up on the device
CMD ["node", "server.js"]
```

Once we have our image ready, we'll write a small Node.JS server that clones the node-bus-pirate repo, runs the tests, makes the results obtainable with an HTTP request, and checks out the latest version of the repo and re-runs the tests on request:

```
var express = require('express');
var app = express();

const Mocha = require('mocha')

let mocha
let failures
```

```
function runTests() {
    mocha = new Mocha()

    mocha.addFile('./hardware-tests/general.js')

    mocha.run((runFailures) => {
        failures = runFailures
    })
}

// Initial run of the tests
runTests()

var simpleGit = require('simple-git')();

// reply to request with 200 if all tests passed, and the
failures if not
app.get('/', function(req, res) {
    if (failures) {
        res.sendStatus(500)
        res.end(failures.toString())
    } else {
        res.sendStatus(200)
        res.end('')
    }
});

// tell our server to pull down changes and run the tests again--
a 500 is sent if the
// pull cannot be completed
```

```
app.get('/update', (req, res) => {
    simpleGit.pull((err) => {
        if (err) {
            res.sendStatus(500)
            res.end(err)
        } else {
            runTests()
            res.sendStatus(200)
            res.end('')
        }
    })
})

//start a server on port 80 and log its start to our console
var server = app.listen(80, function() {

    var port = server.address().port;
    console.log('Example app listening on port ', port);

});
```

Then, we set up our app in Resin! First, we follow the getting started tutorial to set up an app and flash ResinOS to our testing devices. Then, we access our devices' public url and copy it so we can set up our git hook.

Inside the node-bus-pirate repo, we create a file in .git/hooks named post-recieve, and add the following bash script:

```
#!/bin/bash
https://YOUR_DEVICE_URL.resindevice.io/update
```

This means once a push has resolved, the server will recieve a request to pull down the changes and re-run the tests.

Future Improvements

These are things that can be done to further fine-tune this project:

- Add authentication to the test-runner server
- Add a way to only run a subset of tests
- Integrate these results with another CI tools

Hopefully you've enjoyed this walkthrough of the bus-pirate-tester, and it will inspire you to build an automated testing rig of your own.

Thanks for reading!