How Electric Imp Tests Software
Part I: Physical Test Harness

Electric Imp is a powerful platform for developing Internet-connected devices. The gateway to the platform features hardware modules that come in three basic flavours, but each have many features in common including a 32-bit STM32 microcontroller based on ARM's Cortex processor, and a WiFi chip to enable connectivity. The imp001 and imp002 use the F2 version of the STM32, while the imp003 uses the newer F4. The imp operating system includes a virtual machine (VM) which runs byte-code written in the Squirrel programming language. The Electric Imp API provides a Squirrel interface to the STM32 hardware.

Each imp must be taught some WiFi credentials and an Electric Imp account to associate to. The trademarked Electric Imp BlinkUp solution enables end-users to send the imp WiFi credentials, via an optical sensor, so that the imp can connect to our cloud services. Each imp is uniquely paired with another squirrel VM which runs in the cloud. This second VM is known as the imp's *agent*. Communications between the imp, the web and other devices is mediated through the imp-agent mechanism.

Even in this simplistic overview there are several interacting components in our platform, and moreover, it is tightly integrated with the web. The aim of any platform is to provide a flexible, reliable, fast and predictable system for other developers to build on. Testing a platform requires ensuring that all of the API works as documented, while making sure that there are no surprises when creative application developers combine calls to the API in ways that were not envisaged. Perhaps most importantly, any new features or improvements to the Electric Imp platform should not cause existing imps in the wild to stop working. Regression testing is, to use the business vernacular, mission critical.

Apart from the usual and absolutely necessary unit tests in the code, the Electric Imp platform lends itself to three types of larger scale testing. The imp itself is a subsystem with well-defined edges. It talks to a server using a well-defined protocol, and it interfaces with customer's applications via the exposed metal pins. The cloud services also constitute a subsystem with well-defined edges, and uses the same protocol to communicate with an imp, but also communicates with customer's mobile phone apps and the wider web. It is appropriate then to test the imp against a fake server and to test the cloud services against a fake imp and other web endpoints. Additionally, we need genuine end-to-end tests in which a real imp and the real cloud services are tested together as a full system.

In this overview, we will look at the hardware Electric Imp uses to support testing the imp as a subsystem, to ensure it is functioning correctly on every code change. This falls broadly into three sub-categories: first we need a special setup to test that BlinkUp is working; second, when the imp runs some Squirrel code we need to test that the expected things happen at the pins or over the wireless connection; third, we need to test the imp in the face of various types of network error. In a follow-up article we will show how we write the tests themselves, using the best techniques of test-driven development.

BlinkUp and Authentication

In the development process, there are various ways to get code and configuration information on to an imp. We can arrange for the WiFi credentials to be stored permanently on a given test imp before placing it in the test rig. However, a production imp makes use of the Electric Imp BlinkUp technology to receive the relevant credentials. This will be the end-user's first experience with their Electric Imp powered device, and so it is essential that we test that BlinkUp is working reliably on each new code change.

BlinkUp makes use of the built-in optical sensor in each imp. A user enters their Electric Imp credentials and the SSID and password of their WiFi router into a mobile phone app, and the phone's screen then flashes a pattern that the imp can interpret via the optical sensor.

We do not use mobile phones to test BlinkUp, but instead use another imp. The second imp has a simple LED attached to it, which is made to flash in the same patterns that a mobile phone screen would. The test is running on a PC, which is able to send a BlinkUp pattern to the imp, via its agent. The LED is pointing at the optical sensor of the imp under test. To avoid light-pollution, and occasional false negative test results, the two imps are housed in a specially-designed light-proof container. Any resemblance to a shoe-box is purely coincidental. Customer phone apps can send BlinkUp patterns at different frequencies, so we must also test that the imp can receive the pattern across the expected range. The BlinkUp pattern can be sent in either a simple binary form (black/white) or a trilevel form[1], which further increases the testing requirement.

There are different WiFi security protocols, such as Unsecured, WiFi Protected Access (WPA, WPA2) and Wired Equivalent Privacy (WEP). There may also be different encryption methods used with the various protocols. A WiFi router is configured to a particular security protocol, so we use around a dozen different routers to cover all the different flavours. The imp-under-test for BlinkUp is sent each relevant pattern in turn, and we then test that it joins the expected network. This also increases the number of different routers that the imp is continuously tested against.

An imp can also connect to the local wireless network via WiFi Protected Setup (WPS), i.e. by pressing a button on the router, and sending a special BlinkUp pattern to the imp, the imp can autoconnect to the network. Pressing the WPS button can be emulated by sending an HTTP request to the relevant router, so we can test this kind of connection automatically as well.

Connecting the imp to the Network

[1] trilevel BlinkUp is more resilient on phones where the graphics are less reliable.

While it is crucial that the user experience of commissioning an imp is smooth and error-free, it is not the normal operation of the imp. Moreover, testing BlinkUp has different requirements on the test hardware, and where BlinkUp can be tested without any Squirrel code running, most of imp testing involves sending some Squirrel code to the device and ensuring that the imp runs as expected.

The first difficulty is getting the Squirrel to run on the imp. To test a desktop application running on a PC, a sensible approach is to proxy the network connection, and emulate network traffic to that application. This allows for the application to run in a test harness entirely local to the PC. That approach is simply not possible on the imp. The only way to get Squirrel code onto an ordinary imp is for the imp to open a socket to a server, conduct a handshake and receive the code.

For these tests we use a fake server, written in Python. We can instruct this server how to communicate with the imp on a per-test basis, and we can interrogate it for responses from the imp. In particular, we can tell it what Squirrel code to send to the imp. We can also use it as a test-hook: if some of the imp pins are configured for input, we can get the imp to send a message to the fake server to sense the imp's view of what is happening on the pins.

The Hardware

When taken as a subsystem, one edge of the imp is the connection to the Electric Imp cloud services, and another edge is the set of metal input/output pins on the device itself. We provide an API to this hardware in the Squirrel language, and while we curate access to the underlying peripherals, we are as liberal as possible about how the peripherals may be used. It is crucial that code written against the imp API results in the documented behaviour of the imp, and that new features and improvements do not break imps that are already running in the wild. Automated testing is the cornerstone to achieving this. How do we test the imp right up to its edges?

Consider the very simple case in which code is sent from an Electric Imp server to an imp, intending that one of the pins is configured as a simple analogue or digital output. A standard approach to end-to-end testing this functionality would be to set up a circuit to drive, say, an oscilloscope. The Squirrel code for the test could be sent to the imp via the web interface, and the tester could observe the output on the scope. As alluded to above, in an analogous simple case of a single pin being configured as an input, we could set up a circuit to drive the pin, and get the imp to somehow report what it was seeing via the web interface.

But that just covers the simple cases. The imp also has samplers and fixed frequency DACs for recording and playing sound. It has peripherals that require both input and output such as SPI and I2C. The simplistic approach would consign us to ad hoc testing when developing features, and a long tail of testing and bug-fixing before a new imp release. This is not how we want to test our software. We want to know on every code change that the system is still behaving as expected.

This dilemma was unlocked by chance. A member of ARM was visiting Electric Imp's Cambridge office in the United Kingdom, where the imp OS software is developed. He introduced us to the MBED platform which, in short, is intended to help prototype new devices to be built on top of ARM's Cortex technology. The MBED devices have an ARM Cortex microprocessor, and many peripherals exposed through lots of pins.

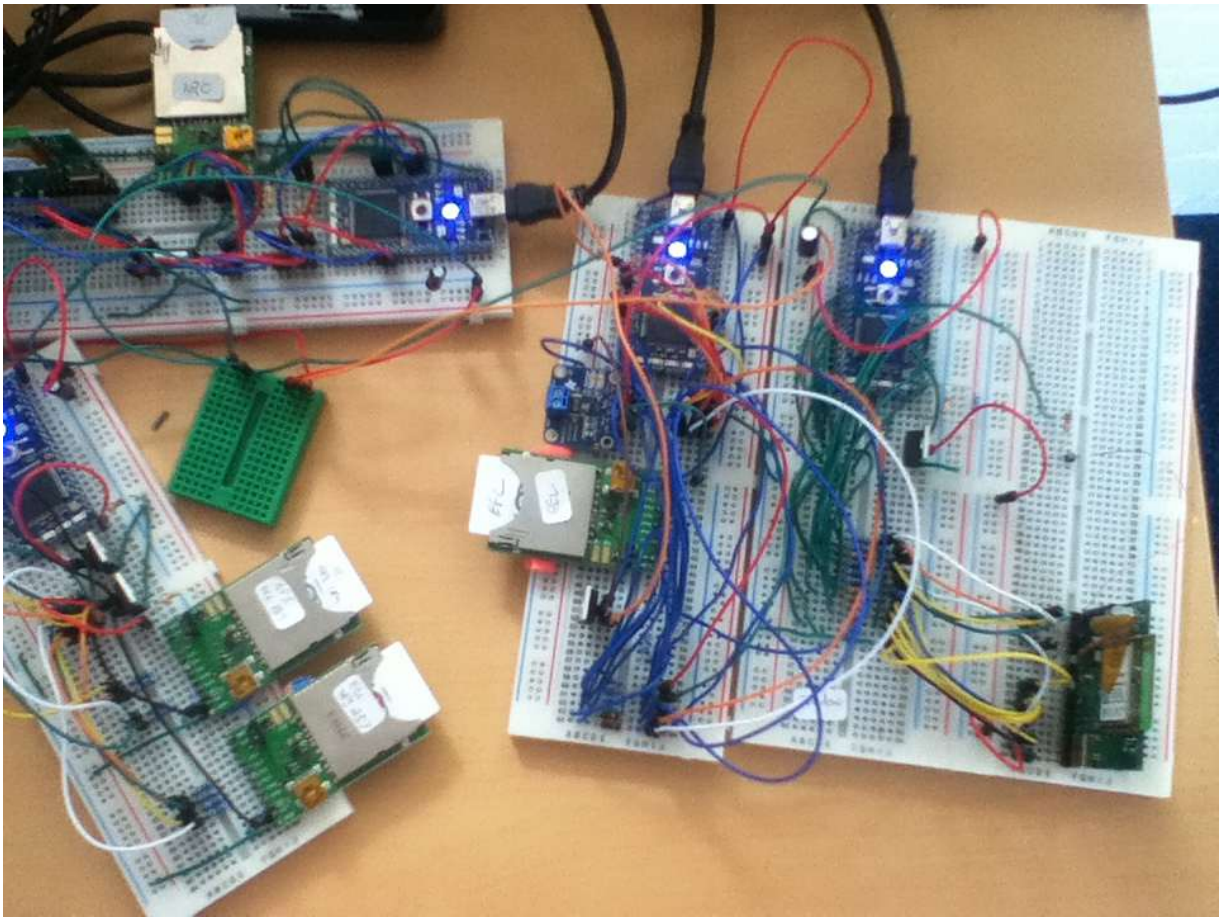
We use the NXP LPC1768 device. MBEDs are designed to be very easy to program. The MBED can be connected to a PC via USB, and it has a filesystem which will appear on the PC. The code that the MBED should run can be copied into this filesystem and it will run when the MBED next boots. It is straightforward to control the power to the MBED since it is drawn over USB from the PC.

Automated Testing

The MBED OS that is shipped with the device exposes a Remote Procedure Call (RPC) interface and provides a Python wrapper for it. This is the glue for automated testing of the imp, since we can have a two-way communication channel between the MBED and the test harness. We run a fake[2] server on a PC, and to the same PC we connect an MBED device. As with the fake server, we can use the Python test harness to both instruct the mbed how to behave, and also to interrogate its view of what is happening on its pins.

The first efforts were made on breadboards. An imp and a PC-connected MBED are slotted in. If the subject of the test is an imp pin configured as an output, then this pin is connected to an MBED pin configured as an input. The fake server, also running on the PC, sends code to the imp which configures and drives the pin to a particular voltage. We then ask the MBED, via its RPC interface, if its pin has detected detected the voltage on the imp's pin. Conversely, we can configure pin on the imp as an input, and the MBED pin as an output, drive the MBED pin using the RPC interface, and have the imp report to the fake server what is happening on its pin.

[2] In the language of test-driven development it is not strictly a fake. A fake tries to emulate the behaviour of the real entity that it is replacing. In part II of this article we will explore the precise nature of the server used for testing.



Now we have all the pieces of the jigsaw puzzle, and we can write a test:

```
class TestDigitalOut(ElectricImpTestCase):
        def test_digital_out_can_write_high(self):
            value = 1
            mbedPin = mbedrpc.DigitalIn(self.mbedrpc, "p10")
            self.loadAndRunSquirrelCode("""
                hardware.pin5.configure(DIGITAL_OUT);
                hardware.pin5.write(%d);
            """ % value
        assert_that(mbedPin.read(), is_(value))
```

Here, we assume that pin p10 on the mbed is connected to pin5 on the imp. The more complicated tests also come relatively easily with the MBED. For example, if we configure the imp such that three of its pins are the MOSI, MISO and CLK (see wikipedia on Serial Peripheral Interface) of an SPI master, then we can connect them to three pins on the MBED, configured as an SPI slave. By extending the MBED OS and RPC interface, we can check what the imp outputs, and control what the MBED sends in response. This is very powerful, and applies just as well to I2C, UART, samplers, fixed frequency DACs and the rest.

This is not a real test, of course, but it is not far off. There is a lot of complexity hidden in here. In the next part we will look at the testing methodology, and in particular how to test the imp in the face of asynchrony.

It is important for these tests to start from a known state, and one part of this is being able to power-cycle an imp. On the early breadboards we achieved this using a field effect transistor (FET), and used one of the MBED pins as a control for the imp power.

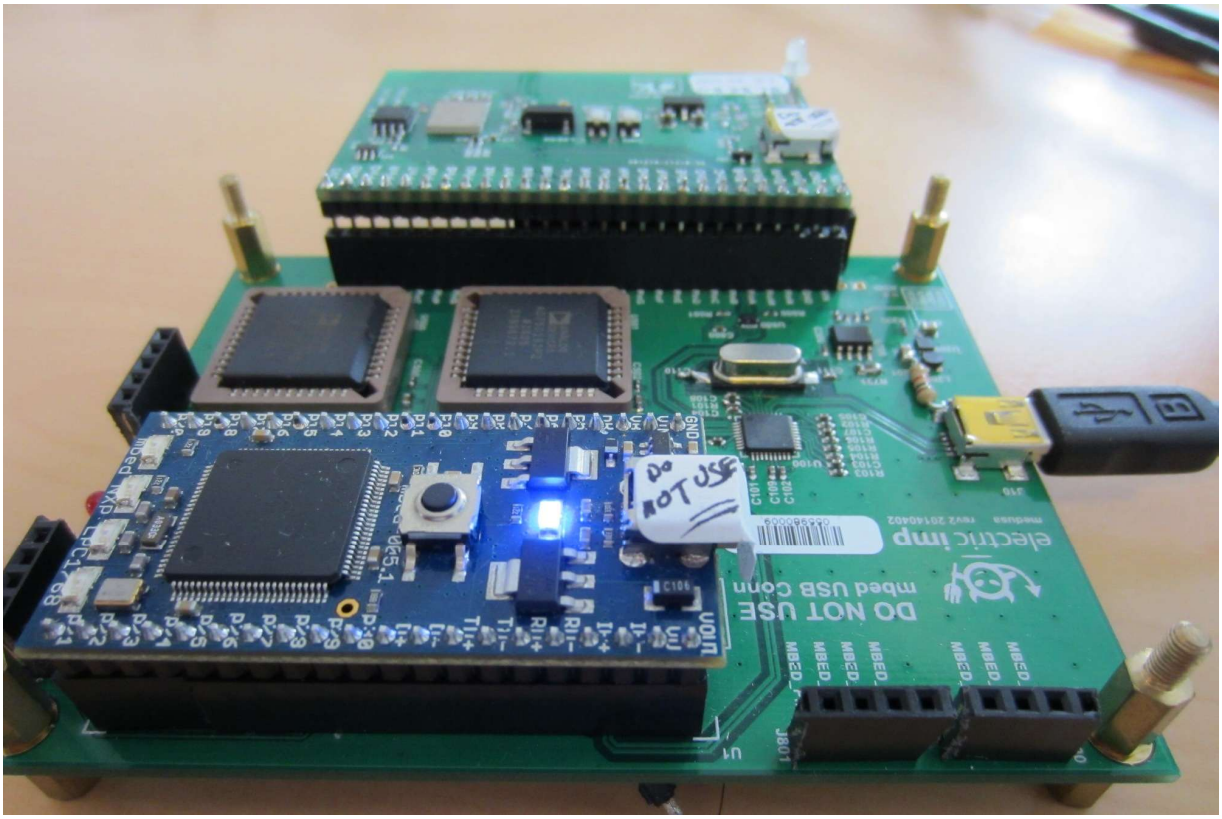
Switching

The LPC1768's peripherals are muxed down onto 20 or so pins, and the imp's peripherals are likewise muxed down on to relatively few pins. The breadboards quickly multiplied. For example, the imp001 and imp002 expose two SPI master peripherals (hardware.spi189 and hardware.spi257). Only one of the MBED SPI peripherals can be properly configured as a slave device that works with all possible configuration of the imp SPI master. We overcame this through some thoughtful wiring of two imps on the same breadboard, choosing two different MBED pins for controlling power to the imps, and arranged for the test suites to be run one after another.

Nevertheless, the breadboard solution does not scale well. Not only are there more and more boards, they are also put together in an ad hoc, test-specific way, meaning that careful circuit diagrams have to be kept for posterity.

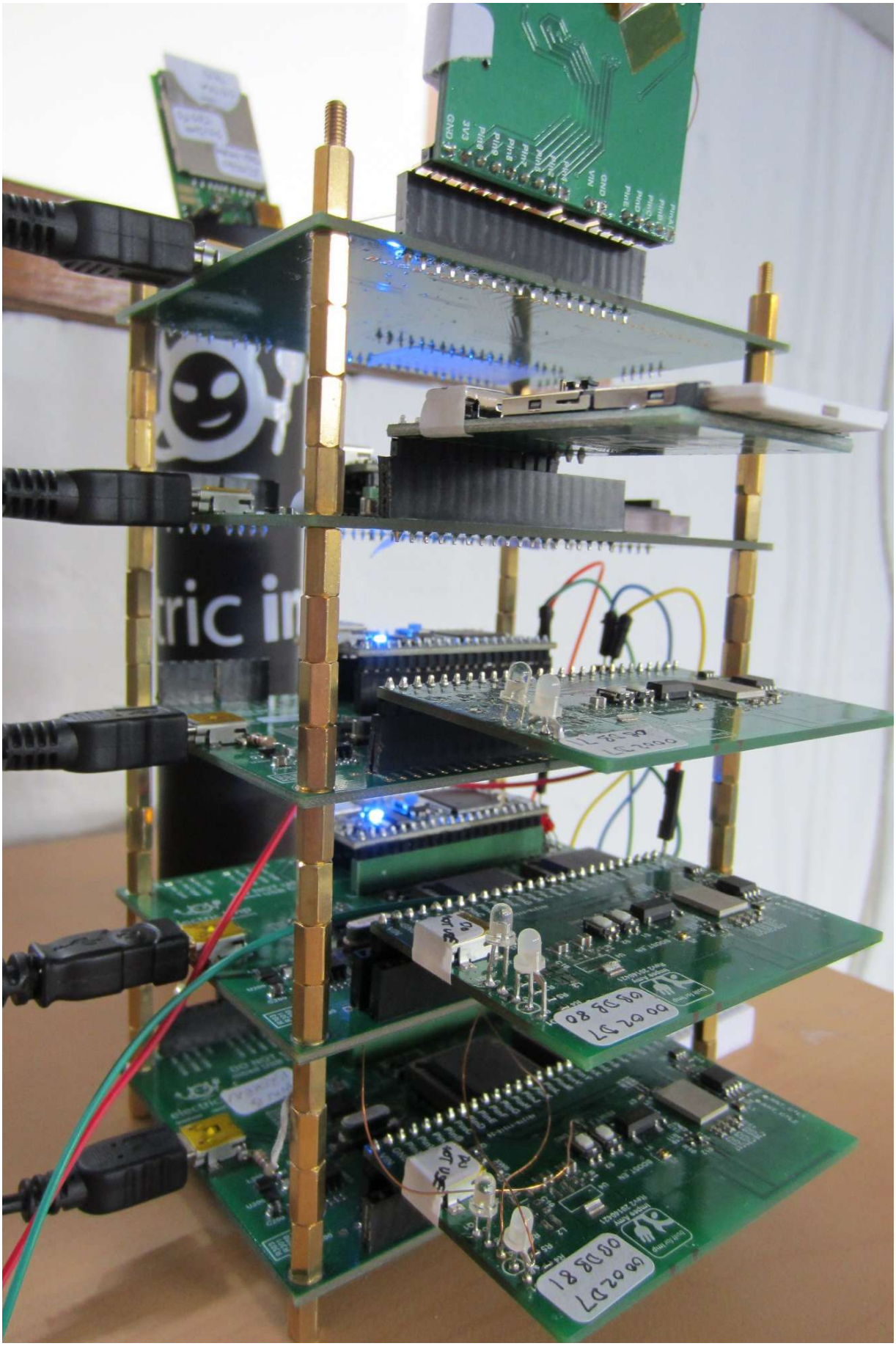
When we started developing the imp003, we also developed a new PCB based on the MBED. In the general case we want to attach any pin of the imp to any pin of the MBED, and we achieve this using an AD75019 16 x 16 Crosspoint switch. The switch is a 16 x 16 square array, and can be configured so that any X terminal can connect to any Y terminal.

Our exceptional hardware team in Los Altos, California took the rough idea and developed a new board, which we call *Medusa*. It contains two Crosspoint switch arrays. An MBED can be plugged in, and there is a clever array of slots so that a single-design Medusa can be used to test different types of imp. The whole rig can be driven by the PC that it is connected to.



To understand the advantage over the original breadboards, consider again the case of testing the imp's SPI master peripherals. Now we test spi189 by programming the switch to connect pins 1, 8 and 9 of the imp to the MBED SPI slave pins and running the test. We then reconfigure the switch to connect pins 2, 5 and 7 to the MBED SPI slave, and run the spi257 tests. Moreover, we are then free to reconfigure the switch to connect, say, an imp I2C peripheral to the MBED I2C peripheral.

The Medusa board is designed to be space-saving, and we can stack them. Here we see one such stack, testing an imp001 in the April development board, an imp002 solder-down module in the Amber development board and three of the new imp003, Murata-manufactured imps in the Amy development board.



Testing Behaviour Against Network Problems

The imp also has well-defined behaviour on network error conditions. Application developers can use the Squirrel API to determine how the imp should behave on unexpected disconnects, e.g. using the function server.onunexpecteddisconnect. Network outages can happen at any point in the route between the imp and the cloud services. It is particularly important that if an unexpected outage occurs during an imp OS update the imp will continue to function correctly, and will still receive the update on resumption of communications with the cloud services.

The imp perceives three different kinds of network disconnections. First, the imp is able to detect a broken connection to the WiFi access point, and when the connection is re-established, the imp is expected to notify the cloud services that there was a local WiFi dropout. We can emulate the WiFi router being switched on or off by sending it an HTTP request from the test harness. Thus, we are able to automatically test that the imp reconnects with the required policy after a local Wifi outage.

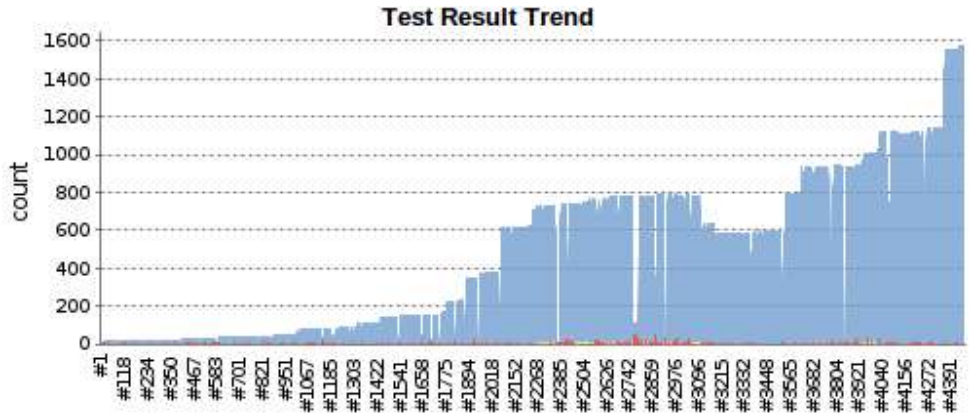
Second, the cloud services may shut down and come back in an orderly fashion, in which case the imp is politely notified of this. Testing the imp's behaviour against server restarts can be done by instructing the fake server to emulate a polite disconnect, and testing that the imp later reconnects with the expected policy.

Third, the network may go away at any point in the route from the access point to the server, and the imp perceives this as an impolite disconnect. Related to this is the possibility of network slowdown, if for example the ISP has rate-limited the traffic from the access point.

The latter type of routing outages are the most challenging to emulate. We achieve it using a Raspberry Pi, which is inserted into the network after the access point, so it can be thought of as a proxy ISP. We communicate with the Pi via HTTP from the test harness, and we can instruct it to allow or deny packets and also to go fast or slow. This test rig runs constantly and causes various network outages to occur during imp OS upgrades, and tests that the imp eventually runs the upgraded OS.

Conclusion

A great side-benefit of the Medusa design is that each developer can have one on their desk and run all of the system tests very easily in their local environment. The tests run automatically on every new commit to the code base. They run repeatedly during idle time on our build PCs. Every week, thousands upon thousands of tests are run against the imp platform. Here is a graph of the history of the imp system test results, as plotted by our jenkins automated build system. The small drop in the number of tests in the middle was due to an optimisation effort, where several things could justifiably be tested by a single test-case rather than several, and the jump near the end is where the new Medusas with the imp003 on were placed into the test rig.



The tests are run in parallel, so, for example, tests on the stack of Medusas shown above run simultaneously. By parallelising the test runs, we are able to run this whole suite of tests in about twenty minutes, which satisfies the strong requirement that every commit to the imp OS code base is fully tested.

The economy of scales is obvious. The same amount of testing done manually would represent weeks or months of effort for the minutes of effort on the automated test rigs. Any bugs found in the manual test cycle would necessarily be much later than code that introduce them, and the cost to an organisation of this effect is by now well-known.

A couple of lucky insights have enabled the Electric Imp engineering team to develop a state of the art physical rig for testing an Internet of Things platform. As the saying goes, the harder you practice, the luckier you get. At the root of the search for this rig is the desire to apply state-of-the art test-driven development techniques from the software world.

That the hardware is so compatible with the software test system is no accident, in the same way that it is no accident that the imp OS development team does not believe in developing features and improvements and expecting the quality of those changes to be checked by a separate team. Lucky insights do not appear to people not looking for them.

In the next article in this series we will look at how we use the best tools of test-driven development to write the tests themselves. We will show how Python’s unittest assert framework, hamcrest matchers and Freeman & Pryce's approach to testing systems in the face of asynchrony can be brought to bear in testing an Internet of Things platform.