



Test-Driving the Internet of Things

**David Dunn
Agile Cambridge 2015**

September 2015

Contents



- 1) **What is Electric Imp?**
- 2) **Unit testing (briefly)**
- 3) **End-to-end testing**
 - a) **History**
 - b) **Python and Hamcrest**
 - c) **Testing an asynchronous system**
- 4) **Summary**

The Electric Imp Connectivity Platform

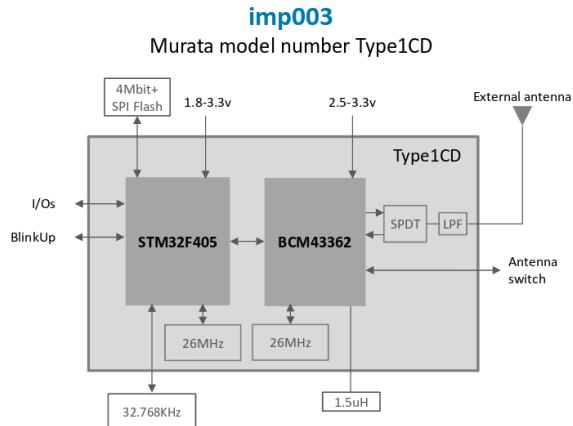


Electric Imp is a connectivity platform

The aim is to let manufacturers do what they're good at - product design and delivery

We take care of connecting devices, in-field updates, security and all that.

Components: the imp module



- Example imp 003.
- STM 32-bit flash microcontroller based on ARM Cortex M processor.
- IOs: GPIOs, UART, SPI, I2C, sampler, fixed frequency DAC etc etc.
- BCM 43362: broadcom wifi chip designed for low power, mobile use. 2.4 GHz WLAN.
- External flash
- Phototransistor for blinkUp
-

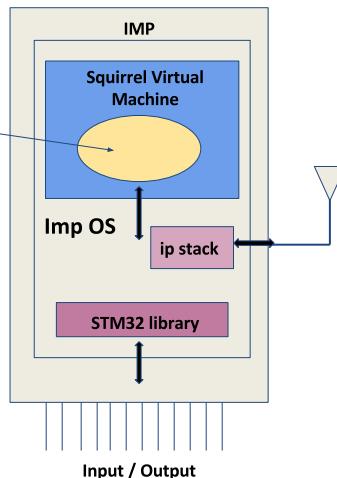
Components: imp operating system

```

174 // This function writes a row of binary image data
175
176 // round width up to next byte boundary
177 local rowbytes = (width + 7) / 8;
178
179 // enforce max width (384 pixels / 8 = 48 bytes
180 local rowbytesclipped = rowbytes > 48 ? 48
181
182 // print
183
184 for (local rowstart = 0; rowstart < height; rowstart++) {
185     local chunkheight = height - rowstart;
186     if (chunkheight > 55) {
187         // put printer in print-binary mode with
188         // a blank line
189         uart.write(0x0d);
190         uart.write(0x0a);
191         uart.write(chunkheight);
192         uart.write(chunkheight);
193     }
194     for (local row = rowstart; row < rowstart + chunkheight; row++) {
195         uart.write(data.readable(rowbytes));
196         if (row == height - 1) {
197             server.log("Done Printing Image");
198         }
199     }
200 }

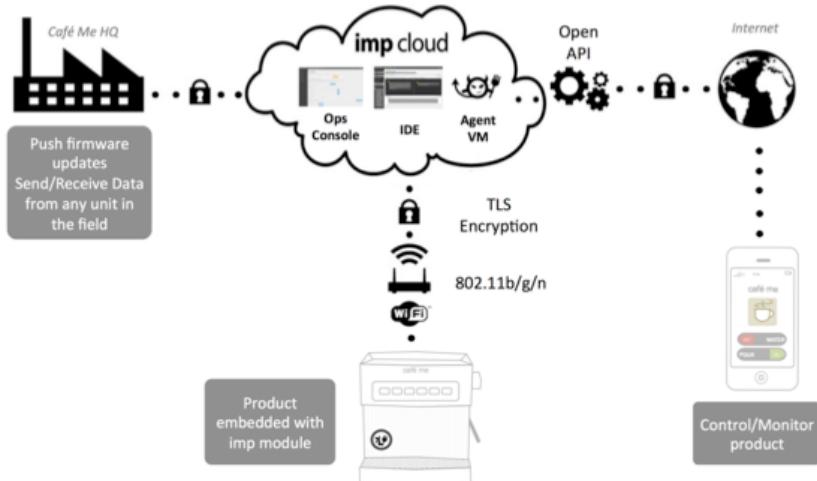
```

Application code



- On the imp, we run an operating system (the imp OS).
- It has an IP stack and wifi drivers for communicating over wifi
- STM32 library to drive the peripherals.
- Runs the Squirrel Virtual Machine for application specific code.

Components: imp cloud



- Each physical imp is paired with another squirrel VM in the cloud.
- AGENT.
- The imp talks to the agent, which then connects to the wider web.
- Agent can provide RESTful endpoints and can talk HTTPS/JSON and so on to other sites.

Components: imp open API



Search...

sampler.configure(*pins*, *sampleRate*, *buffers*, *callback*, *filters*)

Configures the ADC for use

Availability
Device

Returns
Nothing

Parameters

| Name | Type | Description |
|-------------------|---------------------------------------|---|
| <i>pins</i> | pin object or an array of pin objects | Any pin which supports ADC/analog in, or an array of such pins |
| <i>sampleRate</i> | Float or integer | The frequency at which the sampler is set (in Hz) |
| <i>buffers</i> | Array | Array of blobs (up to eight), each a buffer for sampled data. The size of each blob must be a multiple of twice the number of supplied pins |
| <i>callback</i> | Function | A function to be called when a sample buffer is full |
| <i>filters</i> | Constant | Optional pre-processing filters (see below) |

Description

This method configures the sampler. The sampler outputs 16-bit unsigned samples, though with only 12 bits of actual resolution, in the top 12 bits. For example, sampling one pin with a sample rate of 1kHz, 2000 bytes of data are produced every second. With buffers of size 2000 bytes, the [Sampler Callback](#) will be called once per second.

If an array of pins is used, all pins are sampled every period. So when sampling three pins at 1kHz, 6000 bytes of data are produced every second. The pins are sampled almost simultaneously. The output data is interleaved: one sample (two bytes) from the first pin, then one sample from the second pin, and so on, then the second sample from the first

Dev Center

Getting Started

Imp API

- agent
- device
- fixedfrequencydac
- ftp
- hardware
- hash
- http
- httprequest
- httpresponse
- i2c

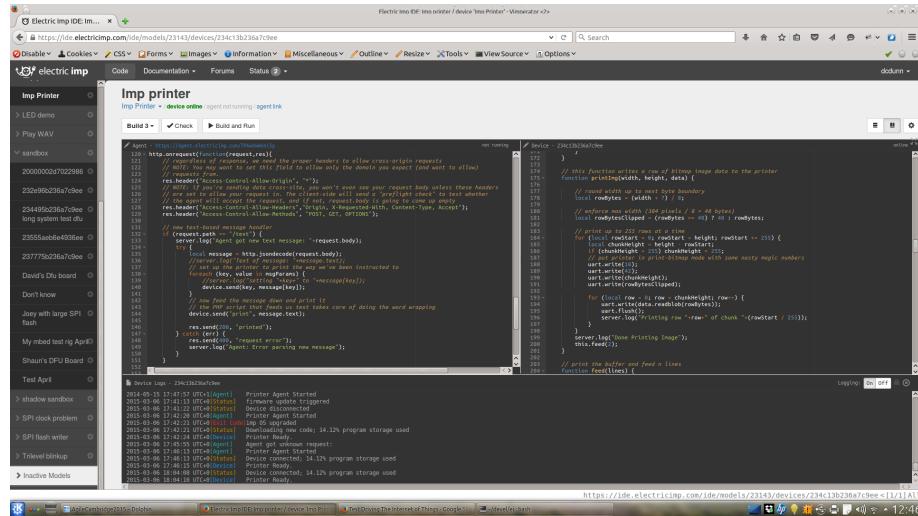
Curated IO.

Components: blinkup



- using BlinkUp we can teach the imp the credential of a local wifi access point.

Components: imp IDE



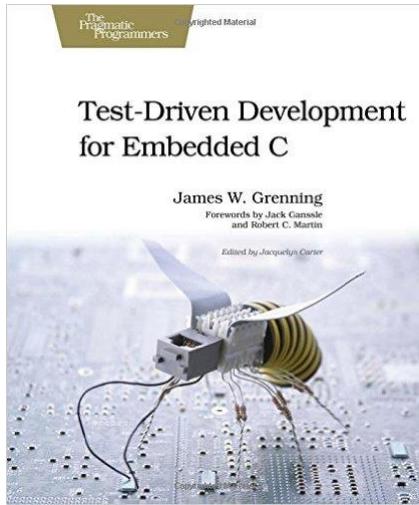
- the application code (imp and agent) is developed in a web browser, compiled and sent over the air to the device.
- a more sophisticated version exists for manufacturers
- e.g. can push code updates to all imps embedded in a given product.

Imp-enabled products



- Hiku: shopping
- Rachio: automated sprinkler. Takes weather forecasts from the web.
- Cybex: smart gym equipment
- Lockitron: smartphone operated door lock.
- EnOcean - monitoring and control systems for automated buildings

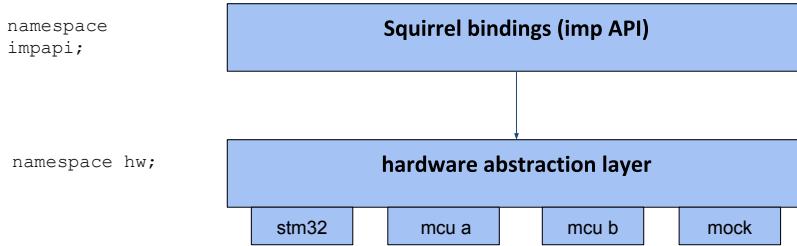
Unit-testing



- **Compile your code to run on a PC as well as the target platform.**
- **We use C++**
- **googlemock**
- **"TDD one instruction away from the hardware"**

- We take Grenning's book as the starting point.
- but use C++, which enables some of the techniques to be achieved more easily using the natural polymorphism of the language.
- googlemock: C++ framework inspired by jMock and hamcrest.

Unit testing: the imp api architecture



- Squirrel layer does parameter checking and passes control to the hw layer
- hw layer interacts with the hardware via third-party libraries.
- Squirrel layer also hands off waitables that can wake up squirrel on a hardware interrupt for example.

Unit-testing squirrel bindings



`spiFlash.write(address, dataSource, writeFlags, startIndex, endIndex)`

Writes a full or partial blob into the SPI flash

Availability
Device (from release 30)

Returns
Integer – 0 if no write verification selected or verification was successful, otherwise a verification failure indicator

Parameters

| Name | Type | Description |
|------------|----------|---|
| address | Integer | Address to write the data to |
| dataSource | Blob | The source blob from which to copy the data |
| writeFlags | Constant | Optional flags to select from pre- and post-verification of the write |
| startIndex | Integer | Optional location within the source blob at which reading starts |
| endIndex | Integer | Optional location within the source blob at which reading stops |

Description

This method will copy the contents of a blob bitwise into the SPI flash at the given address. The rules of NOR flash apply, namely that a write can only program bits (change them from 1 to 0) or leave them alone (see [spiFlash](#)).

The write will succeed on any unprogrammed sectors, but it is not guaranteed to succeed otherwise, since on NOR flash

Dev Center

Getting Started

Imp API

- agent
- device
- fixedfrequencydac
- ftp
- hardware
- hash
- http

- Here is an API for using the external flash that comes with the imp003.

Unit testing squirrel bindings



- The squirrel method `spiflash.write()` is bound to a method on a C++ class

```
SQInteger impapi::SpiFlash::write(HSQUIRRELVM vm);
```

- Objects such as these a) do parameter checking and b) forward to a hardware interface.
- Every bound function has the same signature, and takes a reference to the VM. We have developed a test fixture to simplify getting the VM into the right state for running a unit test.

- to run a unit test requires getting the VM into a particular state, which is not trivial.

Unit-testing squirrel bindings



```
SQInteger SPIFlash::write(HSQUIRRELVM vm)
{
    const int nargs = sq_gettop(vm);
    Variable<int> address(vm, 2);

    if (nargs > 6 || !address.ok || !IsStringOrBlob(vm, 3)) {
        return ThrowBadParameters(vm, "spiflash", "write");
    }

    //... parameter crunching

    m_flashIo->write(address.value, data, bytes);

    return SQ_OK;
}
```

Unit-testing squirrel bindings



```
void testCannotWriteToNegativeAddress()
{
    fixture.compile(
        "hardware.spiflash.enable();\n"
        "hardware.spiflash.write(-3, \"fred\");\n"
    );

    ASSERT_EQ(SQ_ERROR, fixture.run());

    ASSERT_THAT(fixture.messages(), ContainError("address out of range in spiflash.write()"));
}
```

- This shows unit tests for parameter checking: throw an error.

Unit-testing squirrel bindings



```
void testWriteSendsCorrectDataWhenBlob()
{
    const uint32_t expectedAddress = 448*1024 + 1132;
    InSequence sequence;
    EXPECT_CALL(fixture.flash, select());
    EXPECT_CALL(fixture.flash, sendByte(hw::SPIFlash::WriteEnable));
    EXPECT_CALL(fixture.flash, sendByte(hw::SPIFlash::Write));
    EXPECT_CALL(fixture.flash, sendByte((expectedAddress & 0xff0000) >> 16));
    EXPECT_CALL(fixture.flash, sendByte((expectedAddress & 0xff00) >> 8));
    EXPECT_CALL(fixture.flash, sendByte(expectedAddress & 0xff));
    EXPECT_CALL(fixture.flash, write(NotNull(), 4)).With(Args<0,1>(ElementsAre('F', 'r',
'e', 'd')));
    EXPECT_CALL(fixture.flash, unselect());

    fixture.compile(
        "hardware.spiflash.enable();\n"
        "buffer <- blob();\n"
        "buffer.writestring(\"Fred\");\n"
        "hardware.spiflash.write(1132, buffer);\n"
    );
    ASSERT_EQ(SQ_OK, fixture.run());
}
```

- This shows (and documents) how the underlying flash interface must be invoked
- enable for writing
- write command
- 24-bit address in 3 bytes
- the data

Unit testing hw layer



The stm32 implementation of the `hw::SPIFlash` class calls through to the hardware library, through functions such as `SPI_Init`, `SPI_Cmd` and so on.

These cannot be compiled and run on a PC.

Unit-testing hw layer - link-time seams



```
// stm32f2xx_spi.h  
void SPI_Init(SPI_TypeDef*, SPI_InitTypeDef*);  
void SPI_Cmd(SPI_TypeDef*, FunctionalState);
```

Imp image

libstm32.a

Target

unittest.exe

FakeStm32.o

PC

```
// FakeStm32.cpp  
  
void SPI_Init(SPI_TypeDef* spi,  
              SPI_InitTypeDef* initParams)  
{  
    mockSpi->init(spi, initParams);  
}  
  
void SPI_Cmd(SPI_TypeDef* spi,  
             FunctionalState state)  
{  
    mockSpi->cmd(spi, state);  
}
```

One instruction from the hardware

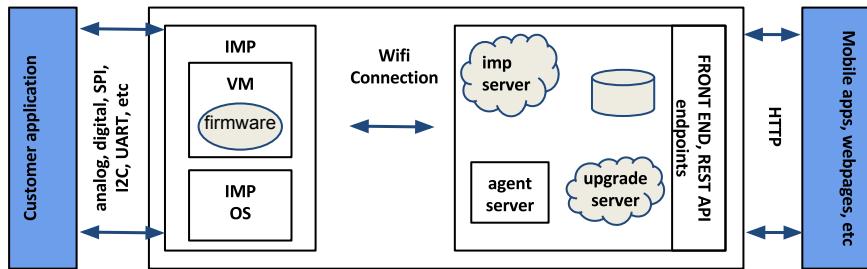
- #include the stm32 header on target and PC.
- Target imp image links the real stm32 library
- The unittest programme on the PC links a faked version of the library
- which uses mocks so we can use the gmock framework.

Unit-testing hw layer



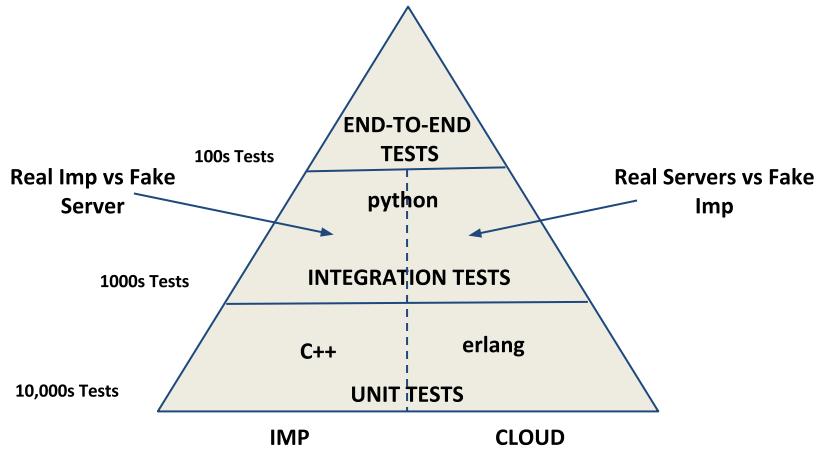
```
void testEnableInitialisesChipSelectPin()
{
    EXPECT_CALL(*fakeSTM32->gpio, init(GpioForLocation(0xAA),
    AllOf(
        GpioMode(GPIO_Mode_OUT),
        GpioOutput(GPIO_OType_PP),
        GpioPull(GPIO_PuPd_NOPULL),
        GpioPinMask(lu << PinForLocation(0xAA)))
    ));
    spiFlash.enable();
}
```

System testing: what is the ‘system’?



- well and good to TDD against the documentation, but that says nothing about the real device
- in one hardware release the manufacturers inverted the sense of the power gates. All unit tests passed. Device dead.

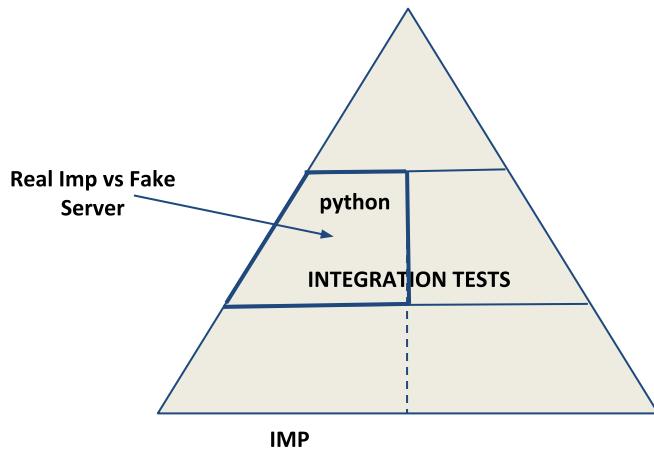
Testing hierarchy



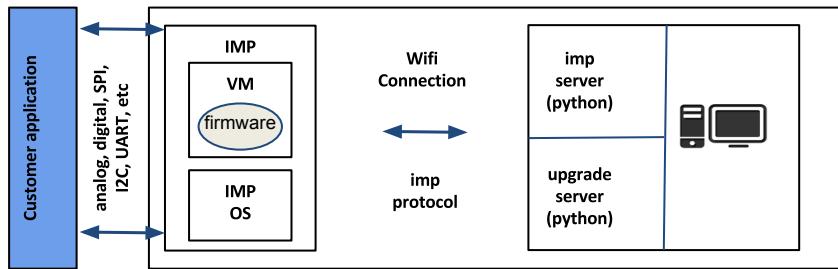
End-to-end hit REST API endpoints, test factory flow, production flow etc

The ends (crudely) are the imps I/O and the REST endpoints.

Testing hierarchy



Faking the cloud



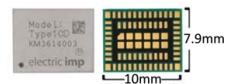
The imp only really sees the imp server and the peripherals, so all the complexity of server can be hidden behind an implementation of a server that speaks imp protocol.

We use iptables, dns forwarding and all that, so that the imp under test THINKS it is talking to the real electric imp cloud services.

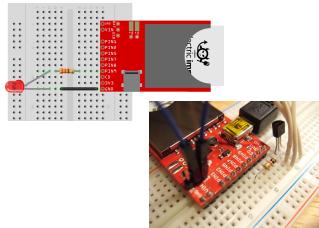
The test space



Factory process
BlinkUp and authentication
LED patterns
Joining Wifi (WEP, WPA, WPS etc)
Communication with imp_server (get squirrel, agent <-> imp comms etc)
imp OS upgrades
Squirrel bindings for peripherals
Non-functional requirements (image size, power consumption, time to boot etc)

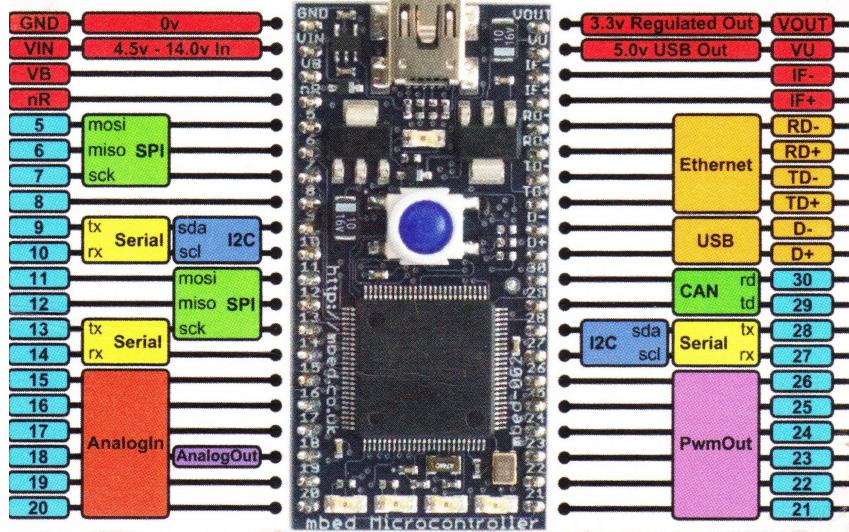


Early efforts: ad hoc application specific testing



Knocking up little applications for targetted testing.
Manually blinking up to lots of acess points.
Whatever.

System testing - the mbed



Hugo Vincent

mbed is development platform for Arm Cortex microprocessor
lots of GPIOs and peripherals

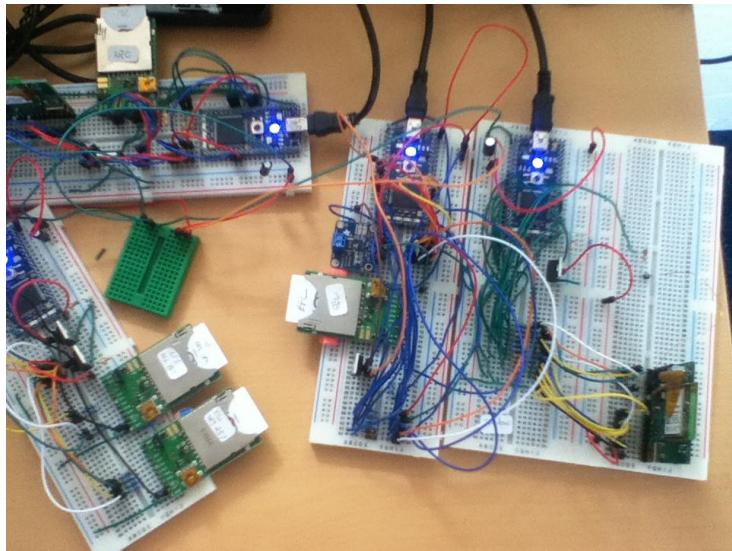
easy to program - connect to PC via USB

python RPC interface to mbed peripherals.

easy to run code on it - when plug in to pc (USB) get a file system
and can copy your binary into it.

has python bindings for its RPC interface

System testing - mbed testing



Breadboards targetting different peripherals.

e.g. GPIOs

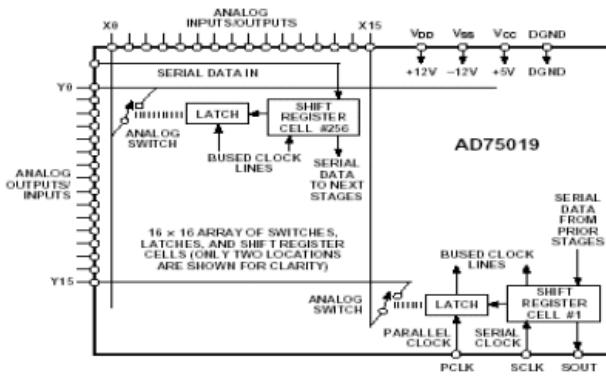
SPI

I2C

BlinkUp

etc

System testing - switching



16x16 crosspoint switch

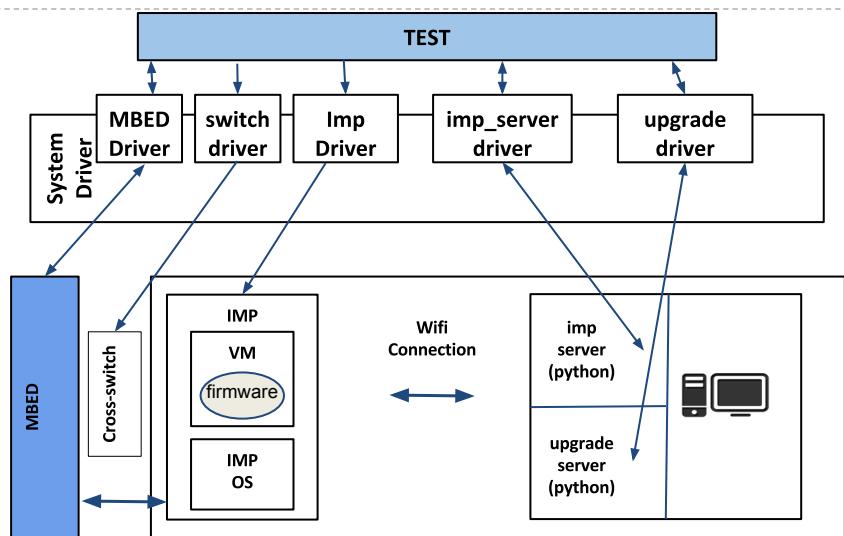
SPI interface to open or close connection between any pair of input/output

System testing - Medusa

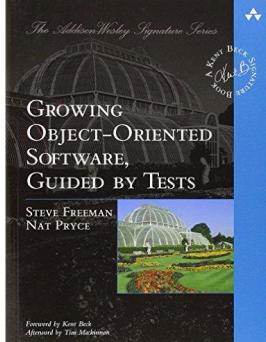
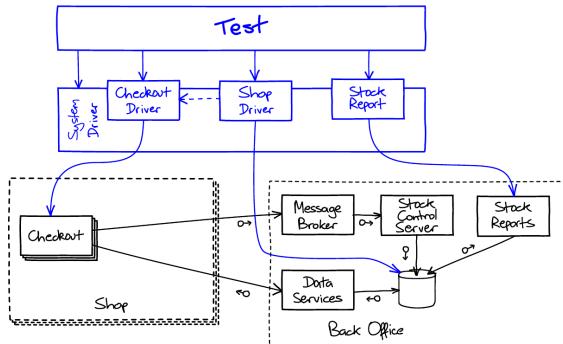


Need two switches to make all imp/mbed connections possible
Two headers so can plug in eel, shadow or joey.

A test harness



Credit where it's due



From "TDD at the System Scale", Agile Cambridge 2010, © Nat Pryce.

- **python**
 - **out-of-the box assert framework, runner, reporter (unittest/nose)**
 - **lots of stable libraries for talking to hardware and the OS**
 - **good thread support for running fake servers etc.**
 - **can run tests at desk on command line**
- **has an implementation of Hamcrest:**
 - **a matcher library for writing easy to read tests**

System testing - a real test



```
class TestExitCodes(ElectricImpTestCase):
    def setUp(self):
        self.imp.power_on()
        self.imp_server.received(handshake())

    def test_exit_code_for_firmware_upgrade(self):
        self.blinkup_imp().at_frequency(60).
            for_firmware_upgrade()
        self.imp_server.received(handshake(),
            last_exit_reason(firmware_upgrade()))
```

This is hiding a lot of stuff - the python imp server, the mbed driver, the switching driver etc are all in the base class.

System testing - another real test



```
class TestDigitalOut(ElectricImpTestCase):
    def setUp(self):
        self.mbedPin = mbed.DigitalIn("p10")
        self.connectImpToMbed("pin5", mbedPin)

    def test_digital_out_can_write_high(self):
        self.loadAndRunSquirrelCode("""
            hardware.pin5.configure(DIGITAL_OUT);
            hardware.pin5.write(1);
        """)

        assert_that(mbedPin, is_high())
```

This is hiding a lot of stuff - the python imp server, the mbed driver, the switching driver etc are all in the base class.

System testing - another real test



```
class TestDigitalOut(ElectricImpTestCase):
    def setUp(self):
        self.mbedPin = mbed.DigitalIn("p10")
        self.connectImpToMbed("pin5", mbedPin)

    def test_digital_out_can_write_high(self):
        self.loadAndRunSquirrelCode("""
            hardware.pin5.configure(DIGITAL_OUT);
            hardware.pin5.write(1);
        """)

        assert_that(mbedPin, is_high())
```

This is hiding a lot of stuff - the python imp server, the mbed driver, the switching driver etc are all in the base class.

System testing - another real test



```
class TestDigitalOut(ElectricImpTestCase):
    def setUp(self):
        self.mbedPin = mbed.DigitalIn("p10")
        self.connectImpToMbed("pin5", mbedPin)

    def test_digital_out_can_write_high(self):
        self.loadAndRunSquirrelCode("""
            hardware.pin5.configure(DIGITAL_OUT);
            hardware.pin5.write(1);
        """)
        sleep(2)
        assert_that(mbedPin, is_high())
```

All you've done is ensure the test takes two seconds longer to run.

Testing in the face of asynchrony



- Shared, thread-safe: updated by the system, examined by the test.
- Snapshot of this state captured atomically by the test and will not change while test is examining it.
- Look for success, not failure. Wait for system to enter expected state.
- Succeed fast. Snapshots can come from polling the system, or can be notified by the system.
 - Pollers
 - Notification traces
- We need all of this for our testing.

System testing - another real test

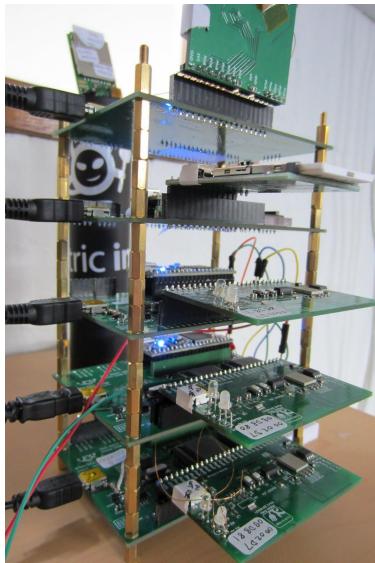


```
class TestDigitalOut(ElectricImpTestCase):
    def setUp(self):
        self.mbedPin = mbed.DigitalIn("p10")
        self.connectImpToMbed("pin5", mbedPin)

    def test_digital_out_can_write_high(self):
        self.loadAndRunSquirrelCode("""
            hardware.pin5.configure(DIGITAL_OUT);
            hardware.pin5.write(1);
        """)
        assertEventually(mbedPin, is_high())
    
```

This is hiding a lot of stuff - the python imp server, the mbed driver, the switching driver etc are all in the base class.

The test harness



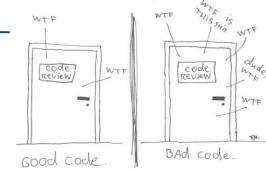
Demo



Workflow



The ONLY VALID MEASUREMENT
OF CODE QUALITY: WTFs/MINUTE



(c) 2008 Focus Shift/O'Sullivan/Holwerda <http://www.osnews.com/comics>