Software

TESTING JERRYSCRIPT NATIVE BINDINGS

Martijn Thé

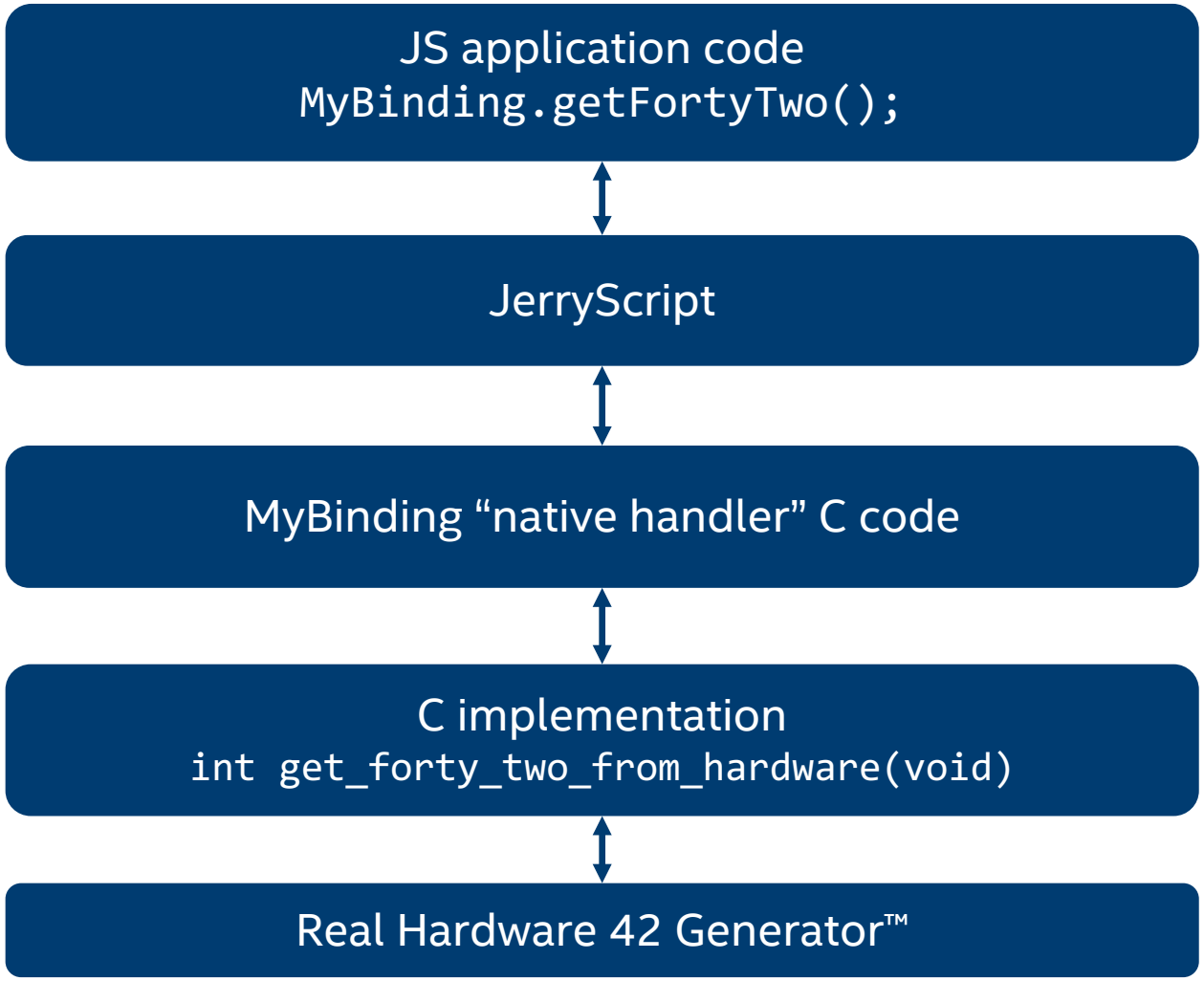
martijn.the@intel.com

Context

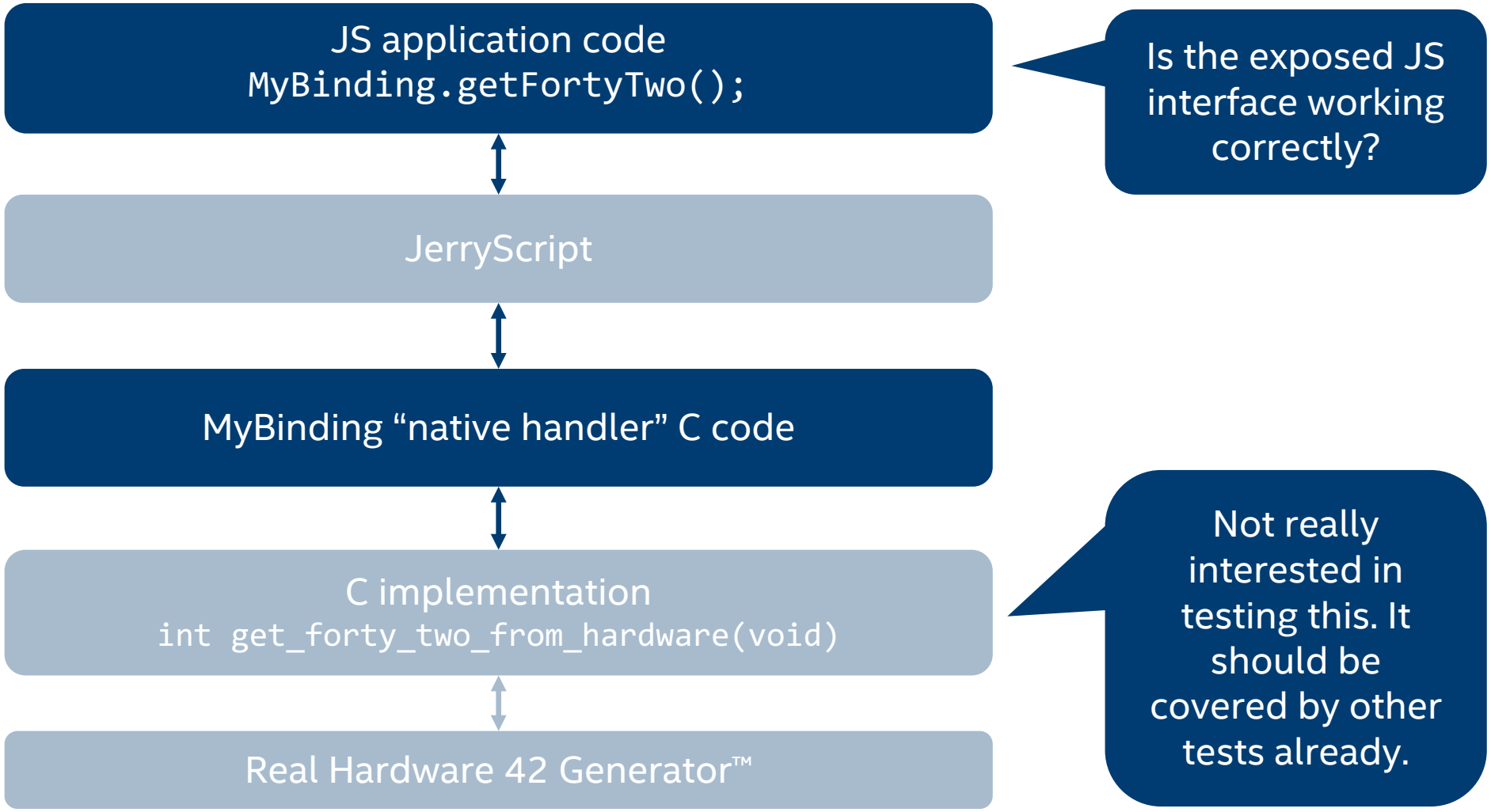
Contrived Example:

```
MyBinding.getFortyTwo();
=>
Binding to access special HW that generates the number 42
```

Context


JS application code
MyBinding.getFortyTwo();
JerryScript
MyBinding “native handler” C code
C implementation
int get_forty_two_from_hardware(void)
Real Hardware 42 Generator™

Context


JS application code
MyBinding.getFortyTwo();
JerryScript
MyBinding “native handler” C code
C implementation
int get_forty_two_from_hardware(void)
Real Hardware 42 Generator™
Is the exposed JS interface working correctly?
Not really interested in testing this. It should be covered by other tests already.

Past approach

- Write (unit) tests in C
- Run them on desktop computer (not on actual target HW)
  - Very fast development cycle
  - Easy debugging
  - Mock/fake/stub out C code that cannot be tested / you are not interested in testing

Past approach

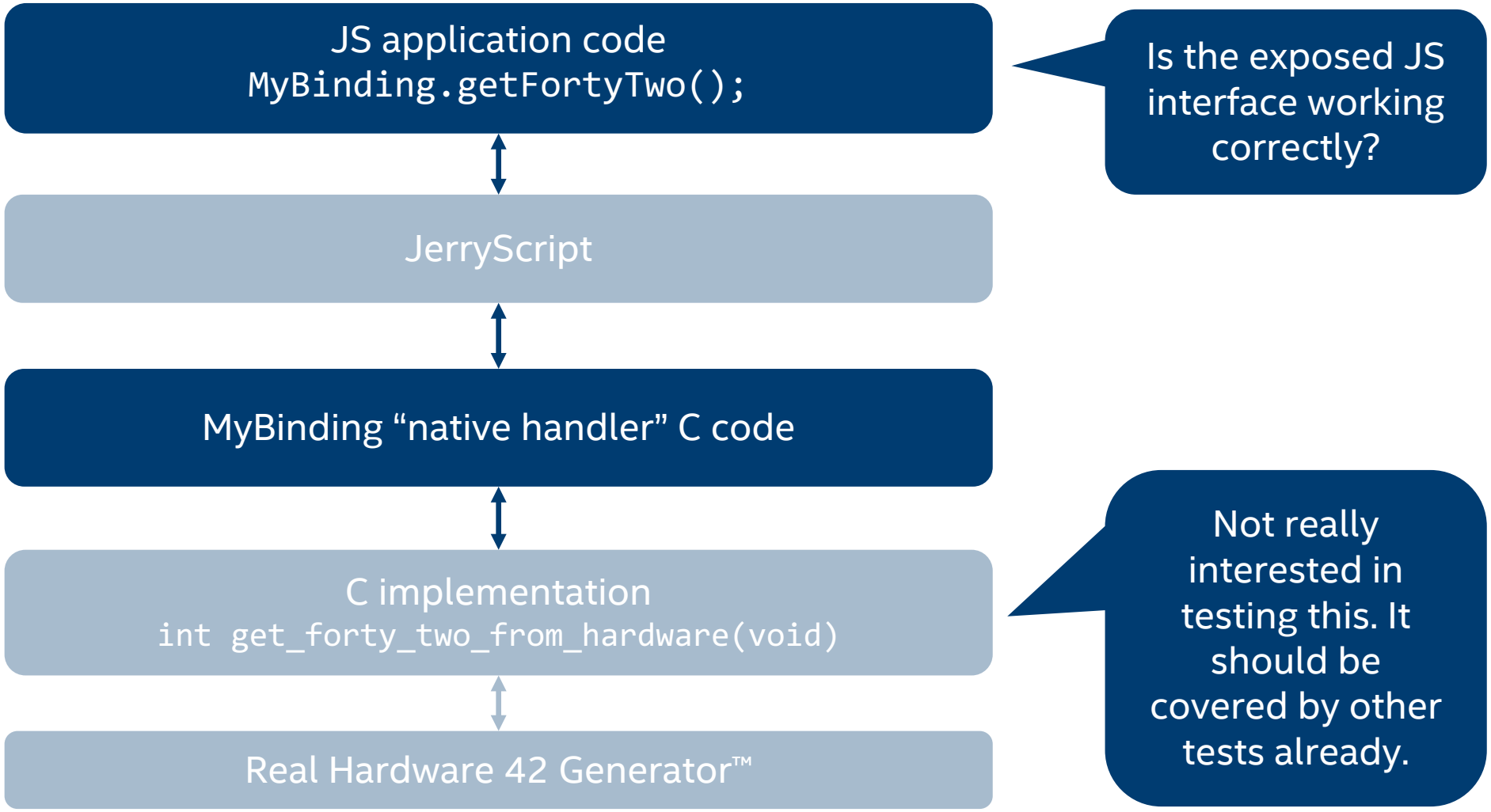
- Simple example, test that myBinding.getFortyTwo() returns number 42
  test_42.c:

```c
void test_mybinding_method_get_forty_two(void) {
        init_jerry_and_load_bindings();
        const char *src = "myBinding.getFortyTwo()";
        const jerry_value_t rv = jerry_eval(
                (const jerry_char_t *)src, strlen(src), false);
        assert(!jerry_value_has_error_flag(rv));
        assert(jerry_value_is_number(rv));
        assert(jerry_get_number_value(rv) == 42);
        deinit_jerry();
}
```

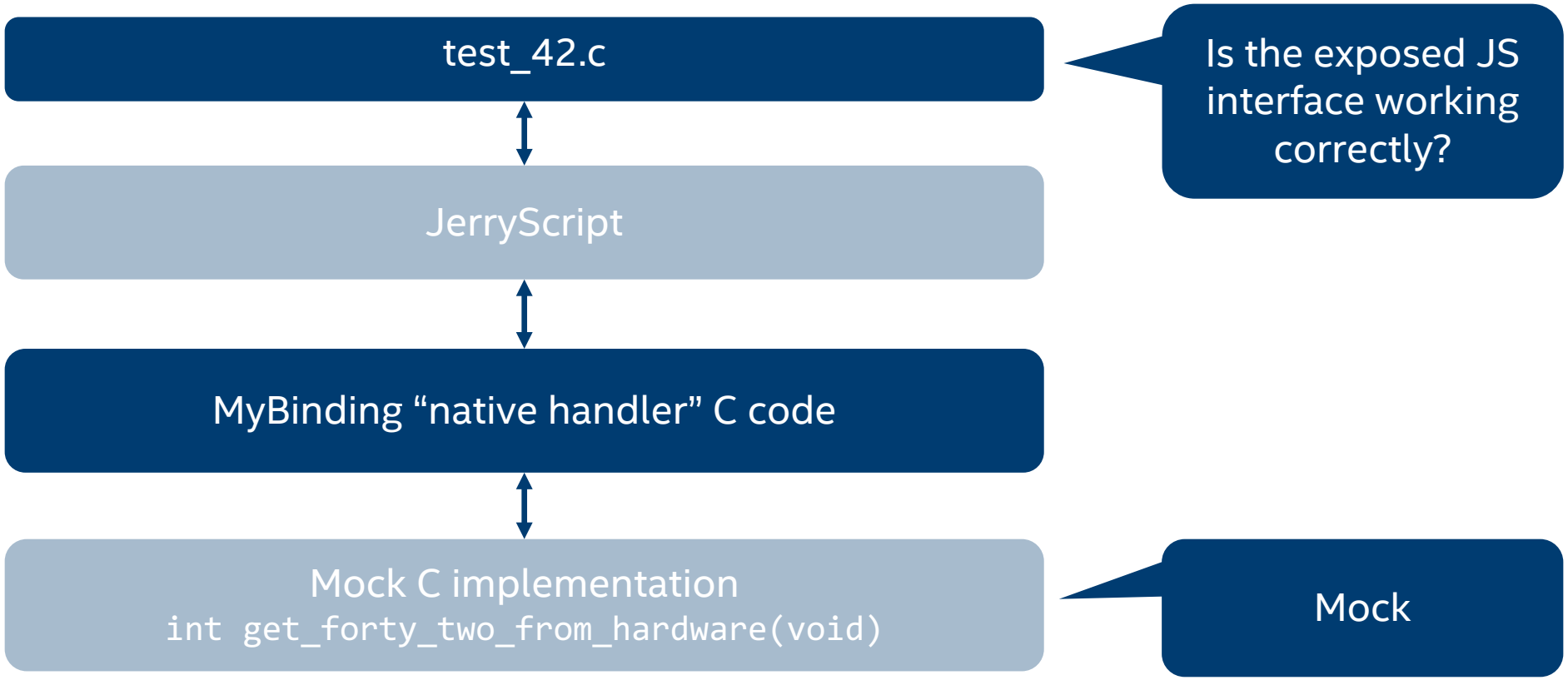
Mocked out driver (special 42 HW is not available on developer's desktop computer):

```c
// mock_42_hw_driver.c
int get_forty_two_from_hardware(void) {
        return 42;
}
```

Past approach



Past approach



Past approach – upsides

- Simple to get started
- Simple to creating mocks for C code that is not under test

Past approach – downsides

- Writing JS test code as C strings is pretty annoying
  - (BTW, the other way around too ;)
- Gets ugly pretty quickly
  - Callbacks, oof…
  - C asserts on more values from JS aside of the eval return value, hrmm…
  - Etc.
- These C test cannot easily be run in other environments, like:
  - Actual target hardware
    - (possible, but requires restructuring mocks/fakes/stubs and creating a test runner)
  - Browser based simulator of your project (no C, perhaps not based on JerryScript, …)
  - Implementation of MyBinding on top of other runtimes, i.e. NodeJS (no JerryScript APIs)

Current approach

- Write binding tests entirely in JS
  test_42.js:

```
test('myBinding.getFortyTwo() works', function() {
  var myBinding = require('mybinding');
  var rv = myBinding.getFortyTwo();
  expect(typeof rv).toBe('number');
  expect(rv).toBe(42);
});
```

Current approach

- “Zest”: Test API *inspired* on Facebook’s Jest
  - https://facebook.github.io/jest/
- Can’t use Jest as-is: too big to run inside JerryScript
- Re-implemented a portion of the Jest API
- Created a JerryScript based runner
  - On top of Criterion C unit test framework for running on desktop computer.
    - Offers: reporting, parallelized running and memory isolation.
    - https://github.com/Snaipe/Criterion/
  - Possible to replace with C unit test runner of choice.

Jest

Search

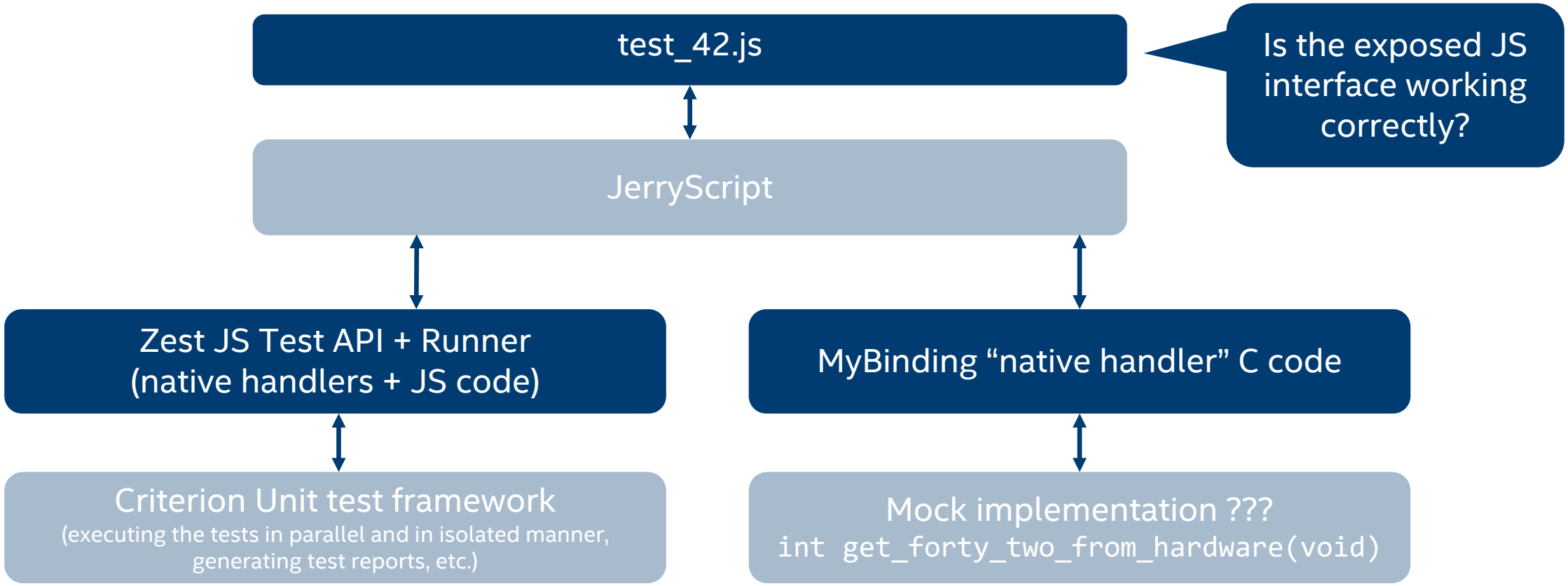
Docs API Help Blog Aあ English GitHub

› API Reference

Methods

- expect(value)
- expect.extend(matchers)
- expect.anything()
- expect.any(constructor)
- expect.arrayContaining(array)
- expect.assertions(number)
- expect.hasAssertions()
- expect.objectContaining(object)
- expect.stringContaining(string)
- expect.stringMatching(regexp)
- expect.addSnapshotSerializer(serializer)
- .not
- .resolves
- .rejects
- .toBe(value)
- .toHaveBeenCalled()
- .toHaveBeenCalledTimes(number)
- .toHaveBeenCalledWith(arg1, arg2, ...)
- .toHaveBeenLastCalledWith(arg1, arg2, ...)
- .toBeCloseTo(number, numDigits)

Current approach



Current approach – upsides

- Simple to get started
- Much faster to write tests compared to previous C approach
- Everything* in JS!
  - * well, almost…
- “Portable” tests
  - Run on development desktop,
  - Run on actual target hardware, etc.
    - In theory at least: plan to run on actual target hardware and simulator, but not done it yet.
  - Run in browser-based simulator,
- Tests themselves are VM-agnostic, test runner is not (yet), but could be added.
  - Or perhaps just run them with Jest + Node.

Current approach – downsides

- Mocks for C code that a JS test needs to control, need to be written as JrS bindings

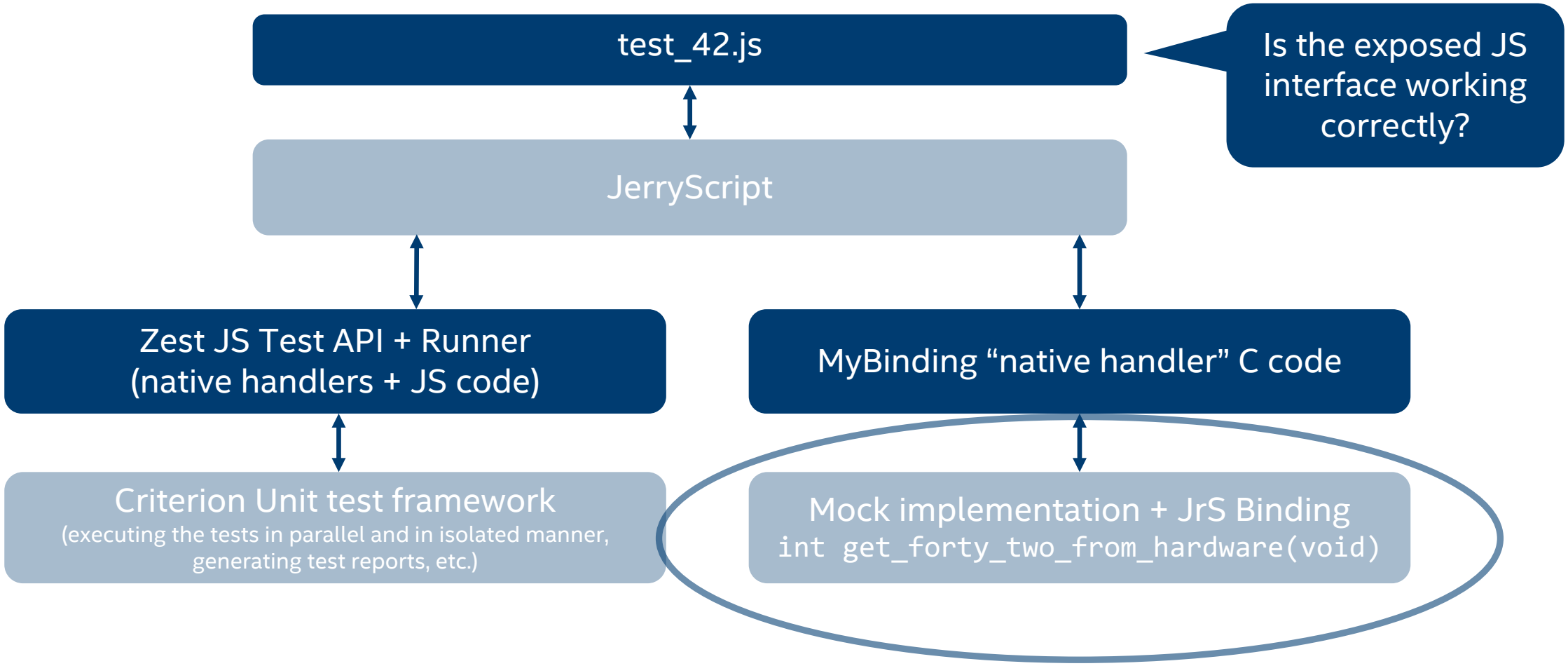
```
test('Magnetometer emits data', function() {
  expect.assertions(3);

  var mockSensor = require('mock_sensor');
  var mag = new Magnetometer();

  mag.start();
  mag.addEventListener('reading', function(e) {
    expect(e.target.x).toBe(123);
    expect(e.target.y).toBe(0);
    expect(e.target.z).toBe(-2);
  });

  mockSensor.emitMagData({
    x: 123.4,
    y: 0,
    z: -2.1
  });
});
```

Current approach



But wait…

- How can a JS test that uses a mock for HW functionality be run on HW?
  - Still TBD, but different solution ideas…

- Depending on the target environment, different mocks/test helpers could be linked:

```
// test_magnetometer.js
test('Magnetometer emit data', function() {
  expect.assertions(3);

  var testSensorHelper = require('test_sensor_helper');
  var mag = new Magnetometer();
  // ...
  testSensorHelper.emit360DegreesClockWise();
});

// env_unittest/test_sensor_helper.js
module.exports.emit360DegreesClockWise = function() {
  var mockSensor = require('mock_sensor');
  for (var deg = 0; deg <= 360; deg += 10) {
    mockSensor.emitMagData({ x: deg, y: 0, z: 0 });
  }
};

// env_device_manual/test_sensor_helper.js
module.exports.emit360DegreesClockWise = function() {
  console.log('Test operator, please rotate the device 360 deg clockwise...');
};

// env_device_auto/test_sensor_helper.js
module.exports.emit360DegreesClockWise = function() {
  var robotArm = require('robot_arm');
  for (var deg = 0; deg <= 360; deg += 10) {
    robotArm.rotateAbsDegrees(deg, 0, 0);
  }
};
```

- Test cases could be annotated with dependencies:

```
// test_magnetometer.js
test.depends({ unittest: true })('Magnetometer emit data', function() {
  expect.assertions(3);

  var testSensorHelper = require('test_sensor_helper');
  var mag = new Magnetometer();
  // ...
  var mockSensor = require('mock_sensor');
  for (var deg = 0; deg <= 360; deg += 10) {
    mockSensor.emitMagData({ x: deg, y: 0, z: 0 });
  }
});

test.depends({ robot: true })('Magnetometer emit data', function() {
  expect.assertions(3);

  var testSensorHelper = require('test_sensor_helper');
  var mag = new Magnetometer();
  // ...
  var robotArm = require('robot_arm');
  for (var deg = 0; deg <= 360; deg += 10) {
    robotArm.rotateAbsDegrees(deg, 0, 0);
  }
});
```

- Tests could perhaps query the capabilities at runtime:

```javascript
// test_magnetometer.js
test.depends({ unittest: true })('Magnetometer emit data', function() {
  expect.assertions(3);

  var testSensorHelper = require('test_sensor_helper');
  var mag = new Magnetometer();
  // ...
  if (test.capabilities.unittest === true) {
    var mockSensor = require('mock_sensor');
    for (var deg = 0; deg <= 360; deg += 10) {
      mockSensor.emitMagData({ x: deg, y: 0, z: 0 });
    }
  } else if (test.capabilities.robot === true) {
    var robotArm = require('robot_arm');
    for (var deg = 0; deg <= 360; deg += 10) {
      robotArm.rotateAbsDegrees(deg, 0, 0);
    }
  }
});
```

finally {

- We've written +300 JS test cases for our internal project
  - Happy that we took this route
- Looking forward to get input on concepts and other approaches to testing
- Not open source at the moment, but if valuable to others we could open it up

Q&A