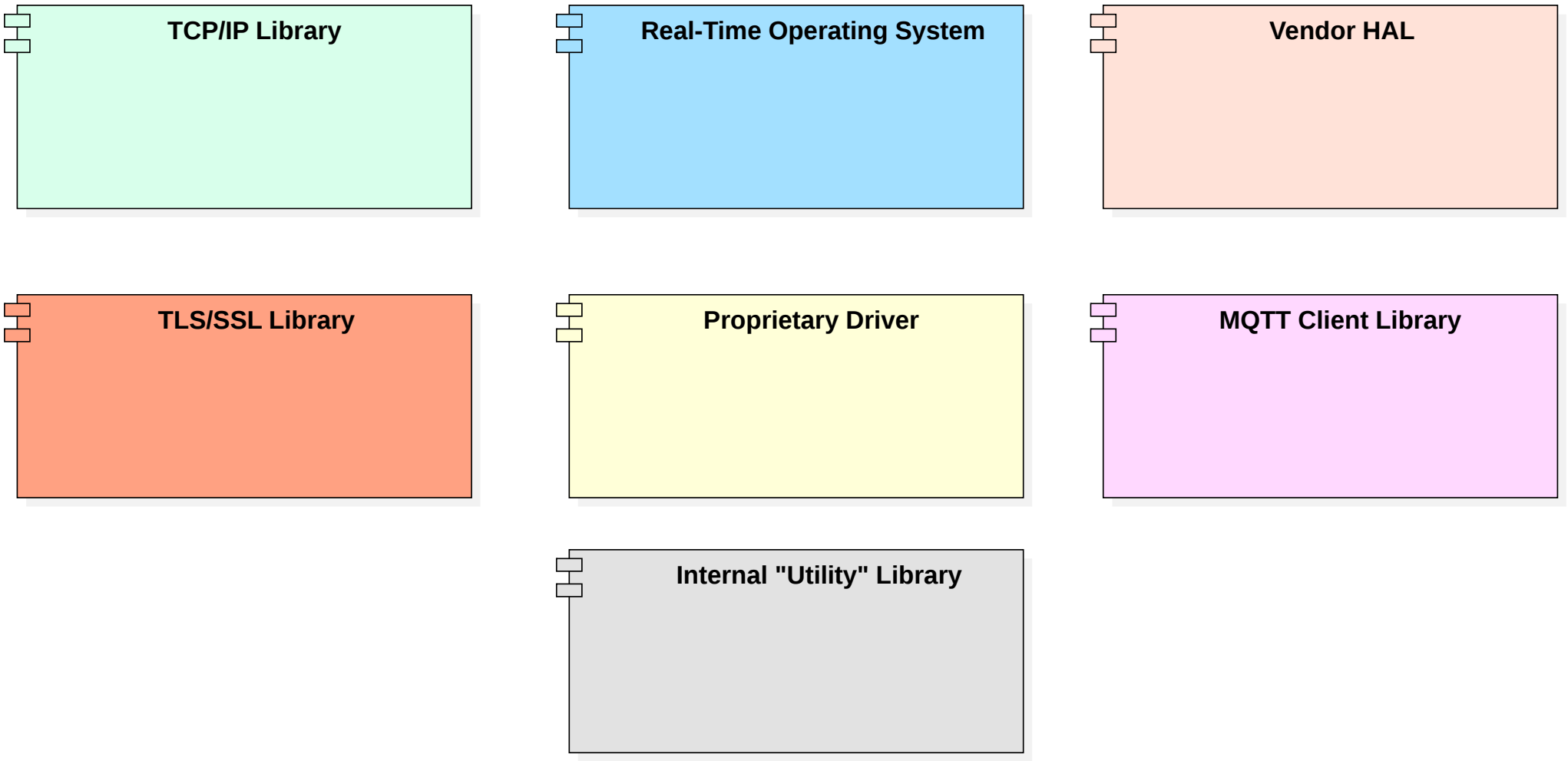
Microcontrollers in Micro-Increments

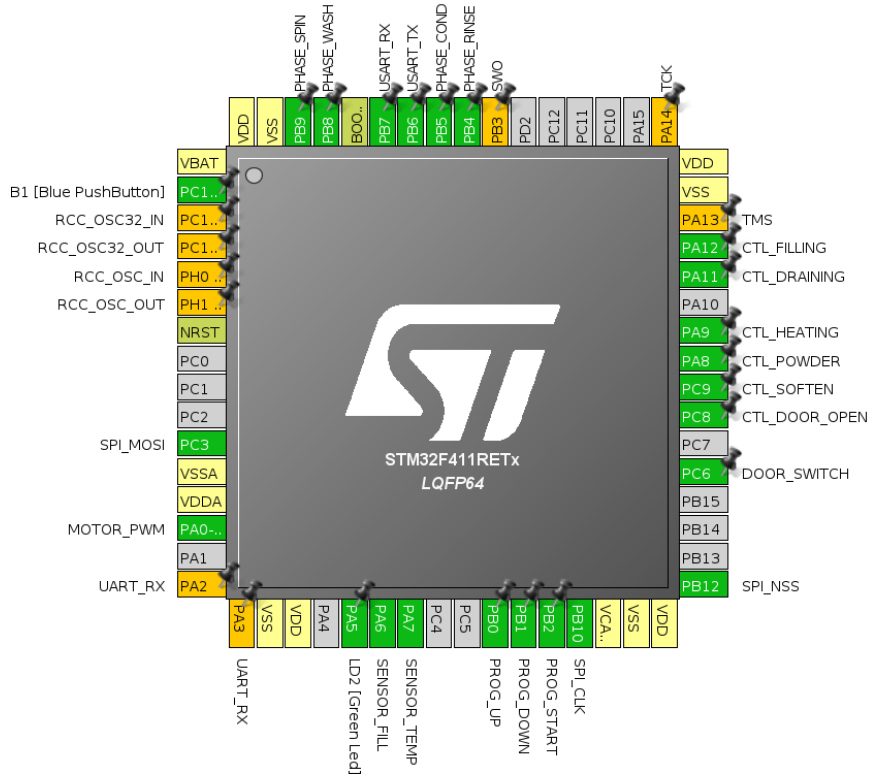
A Test-Driven C++ Workflow for Embedded Systems

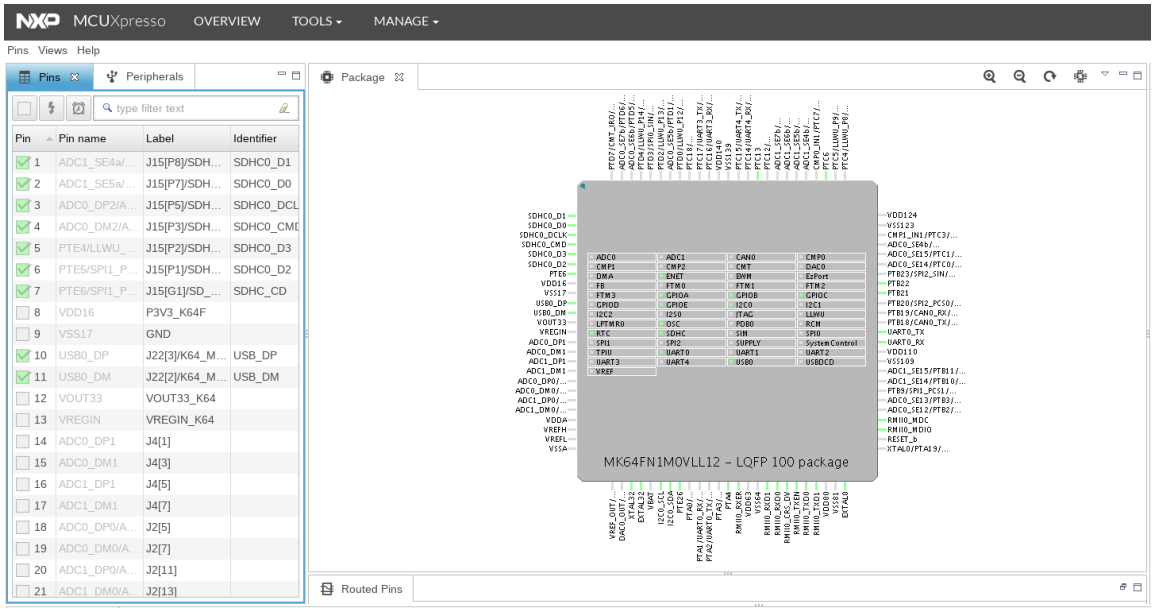
Mike Ritchie
Developer, Coach, Builder of Dubious Hardware

Welcome to your new “greenfield” project



Pin configuration & code generation





```
43
44 /* USER CODE BEGIN 1 */
45
46
47 /* USER CODE END 1 */
48
```

James Grenning
@jwgrenning

Following

A clean embedded architecture is a testable embedded architecture.

8:52 pm - 2 Apr 2017

6 Retweets **10** Likes

1 6 10

Why TDD ? Stop me if you've heard this...

- Gives you "in the moment" feedback on your design
- You're a user of your code before you write it
- Leads naturally to decoupled, testable units
- Gives you a safety net for refactoring confidently
- Oh, and when you're done...you actually have tests

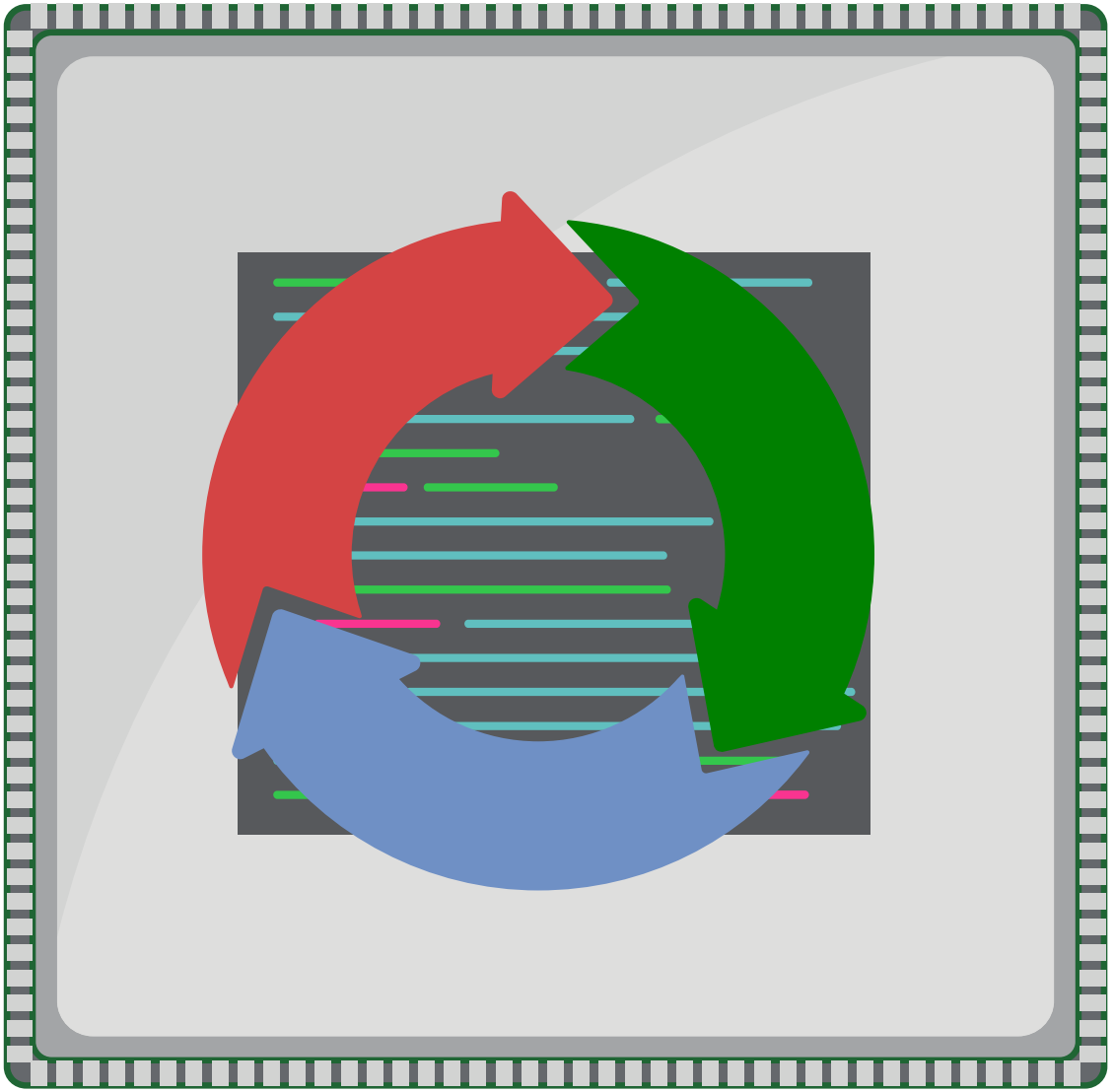
Why TDD for embedded ?

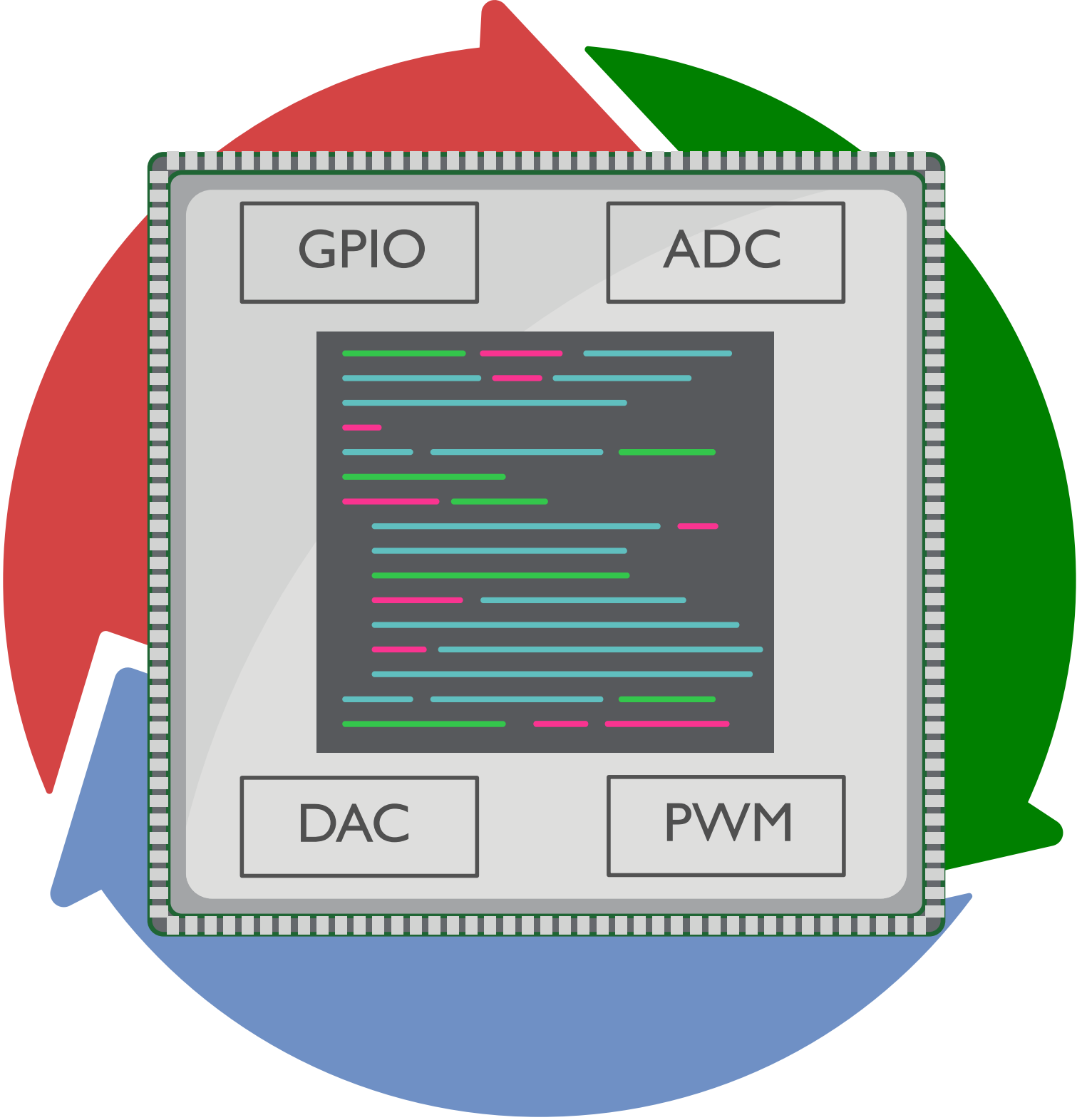
The blankest of canvasses, and complex requirements

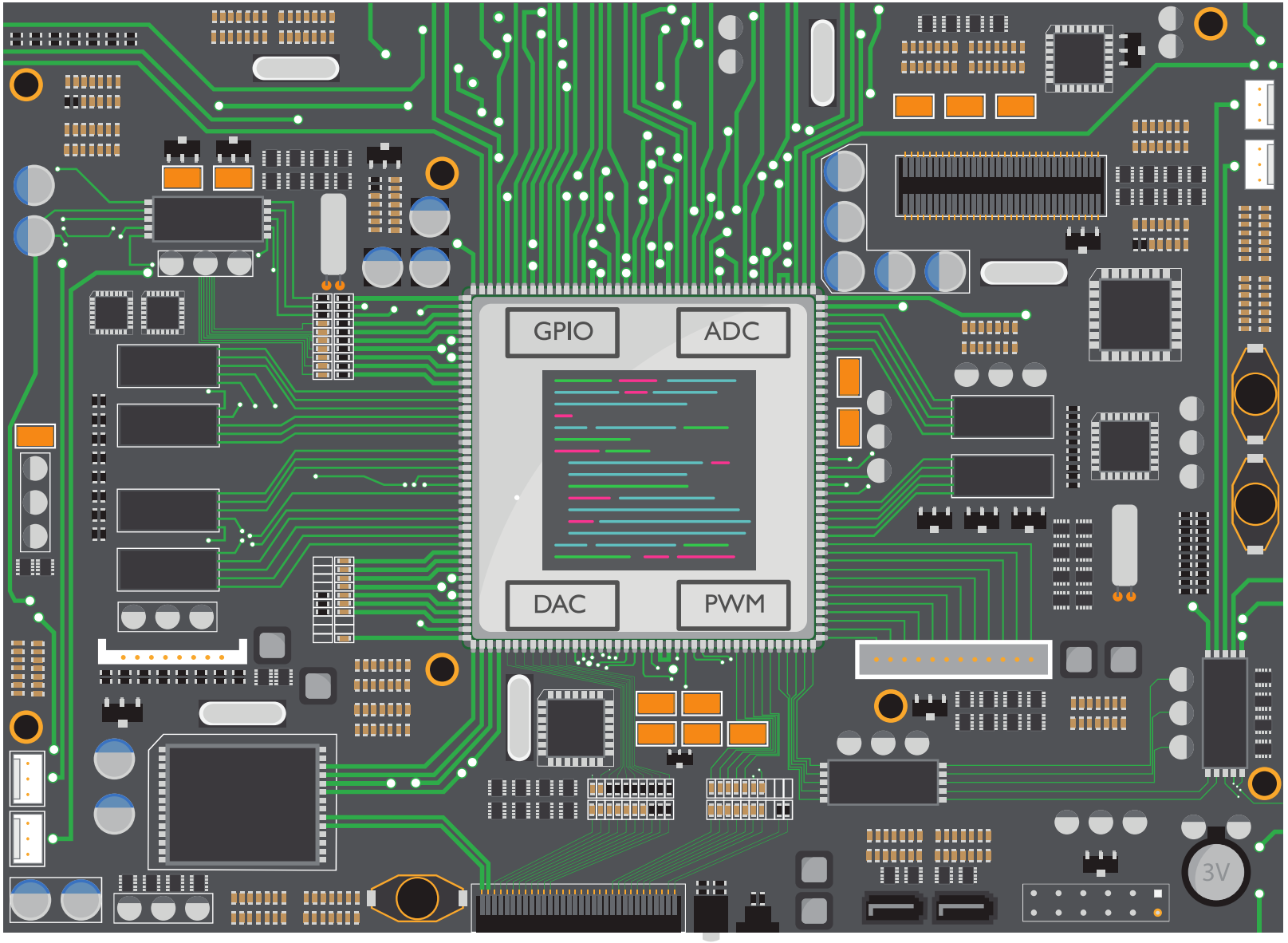
Tight constraints and unforgiving platforms

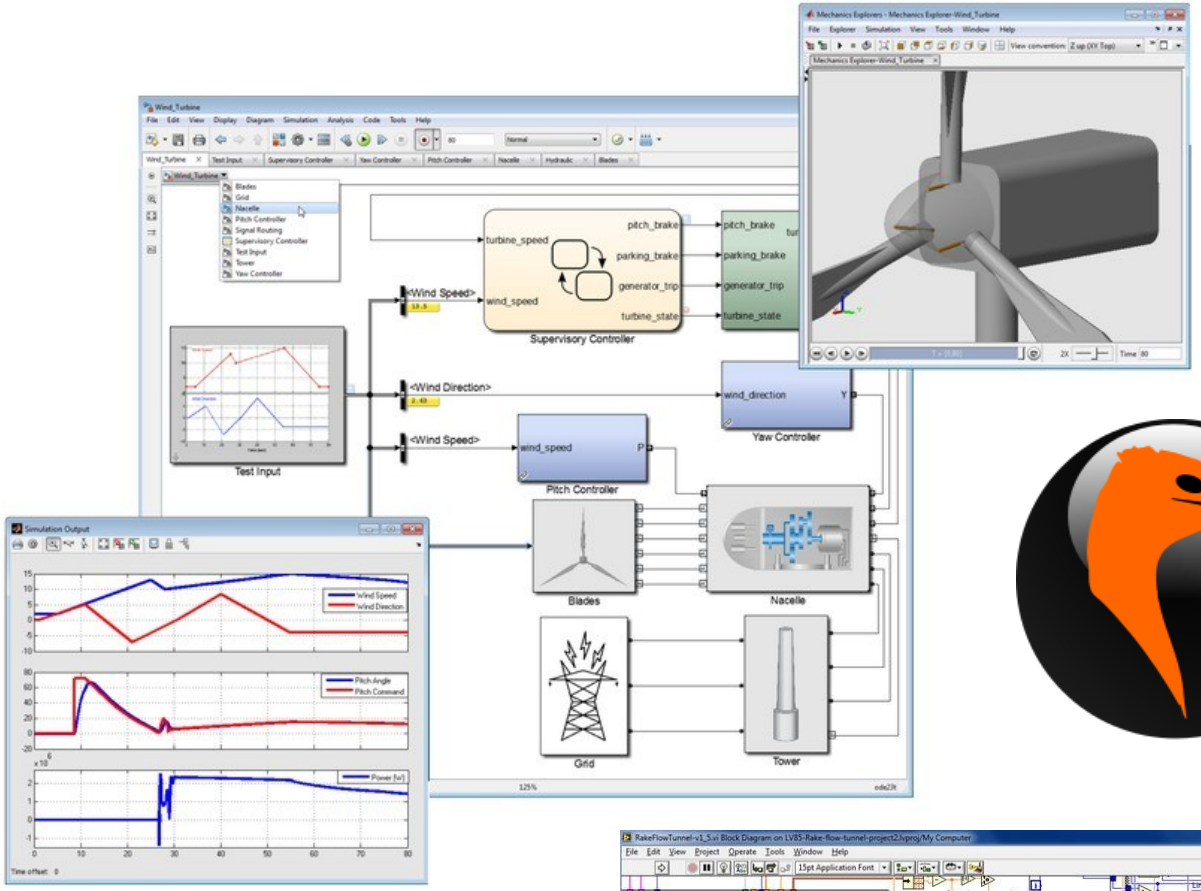
Inherent complexity of cross-build and debug

Expectations of high levels of safety and reliability



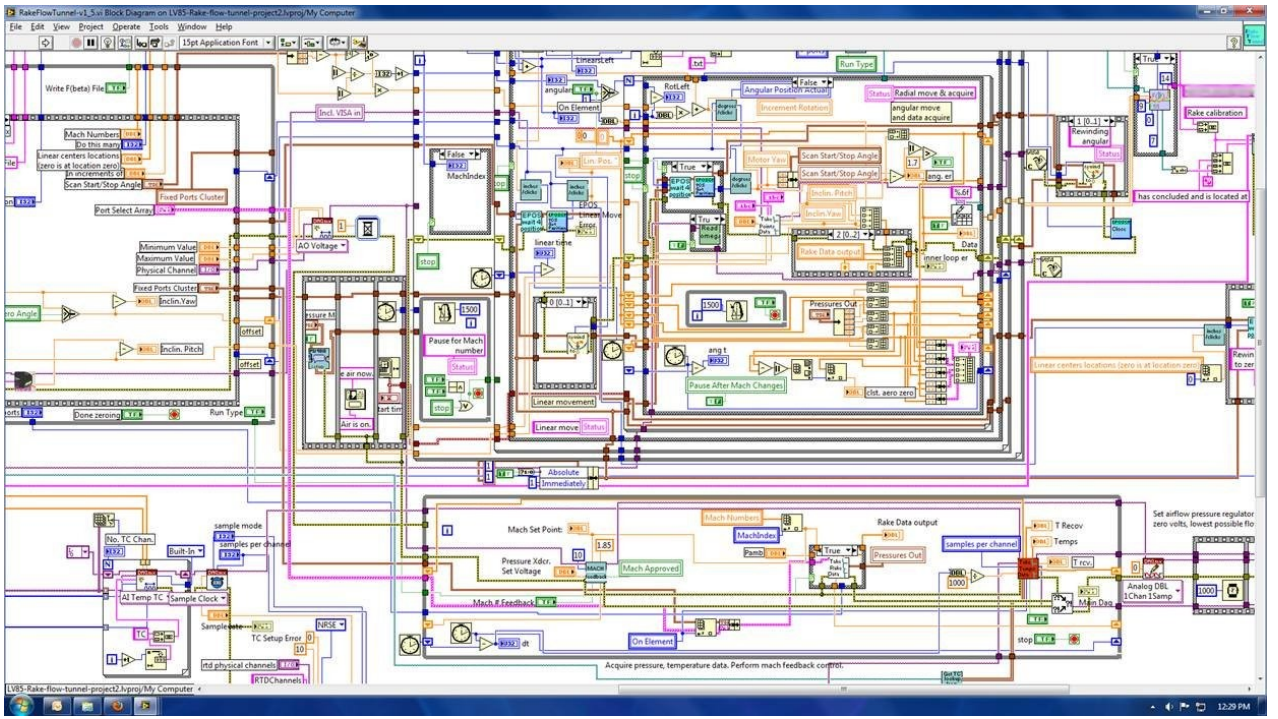

GPIO
ADC
DAC
PWM


GPIO
ADC
DAC
PWM
3V



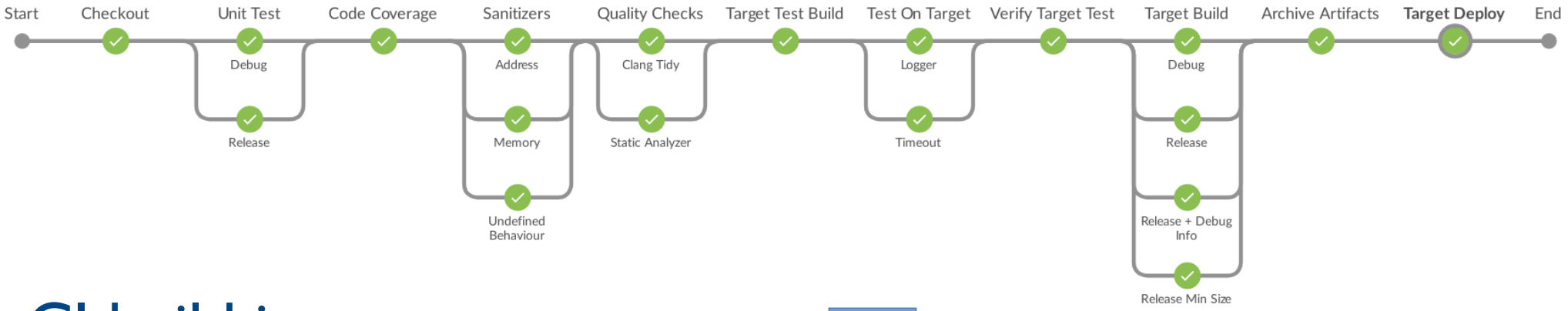
Simulink image CC-BY-SA Wikipedia Mcarone1

QEMU

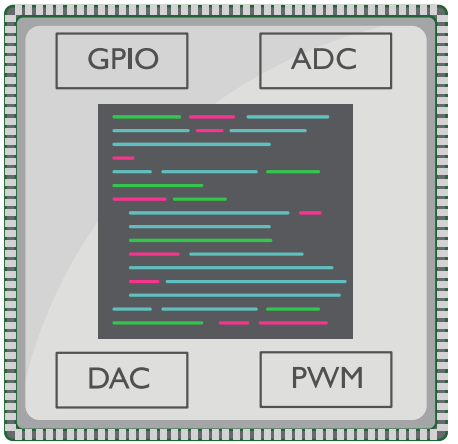


LabVIEW image (c) National Instruments

Fast and fluent TDD on the host, getting near-instantaneous feedback on changes.



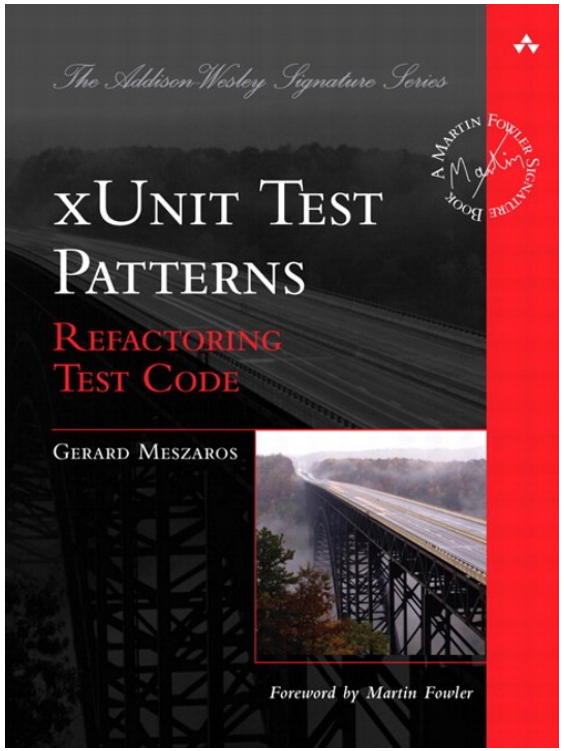
CI build is run on every commit, builds are deployed to evaluation/target hardware for test-on-target and subsequent profiling and analysis.



Patterns for testability

Doubles, Stubs, Mocks, Spies

- Central to testability is substitutability of implementation
- Various forms of test double help us to isolate code for test
- People don't always agree on definitions...or even usage!



We're going to use a few techniques in this session. For a comprehensive view of test patterns, this book by Gerard Meszaros is the go-to reference work.

What's your separation of concerns ?

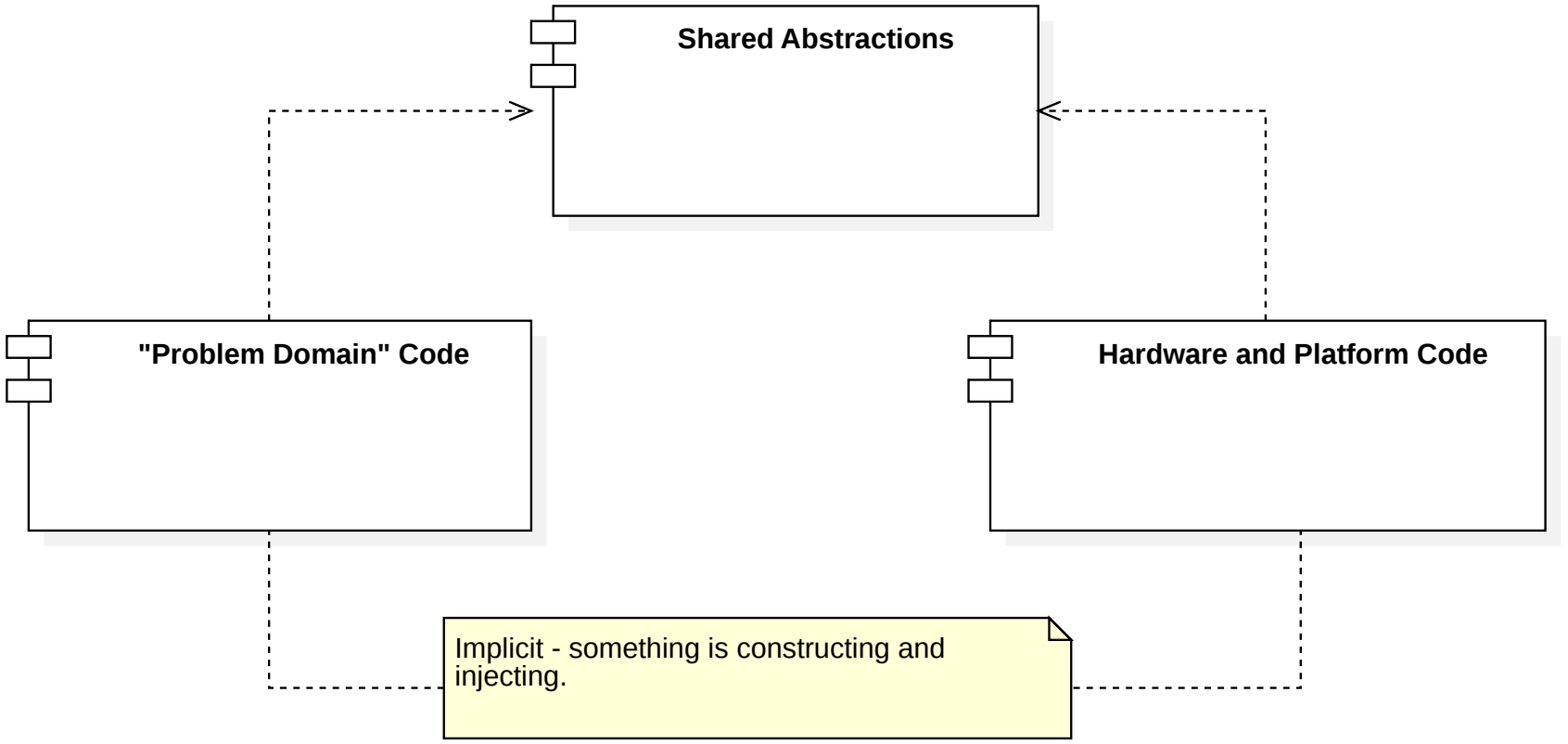
Interacting with an RTOS ?

Core problem domain logic ?

Interacting with hardware ?

Networking ?
Crypto ?
etc.

Dependency Inversion : the biggest hammer



"[A] High level modules should not depend upon low level modules. Both should depend upon abstractions. [B] Abstractions should not depend upon details. Details should depend upon abstractions."

Dependency Inversion : the biggest hammer

```
struct indicator_led {
  virtual ~indicator_led() = default;
  virtual void on() = 0;
  virtual void off() = 0;
};

struct washing_machine {
  virtual ~washing_machine() = default;
  virtual void door_opened() = 0;
  virtual void door_closed() = 0;
};

class washer : public washing_machine {
public:
  explicit washer(indicator_led &indicator);
  void door_opened() final;
  void door_closed() final;

private:
  indicator_led &indicator;
};
```

TDD with dependency-inverted mocks

```
struct mock_indicator : public indicator_led {
  MAKE_MOCK0(on, void());
  MAKE_MOCK0(off, void());
};

TEST_CASE("The washing machine") {

  mock_indicator indicator;
  washer washer{indicator};

  SECTION("When the door is open, illuminates door LED") {
    REQUIRE_CALL(indicator, on());

    washer.door_opened();
  }
}
```

What really is a “shared abstraction” though?

```
template <typename T> class water_system {
public:
  explicit water_system(T &valve)
      : water_fill_valve{valve} {}

  void handle_sensor_reading(uint32_t water_level) {

    if (water_level < fill_target) {
      water_fill_valve.open();
    }
  }

private:
  T &water_fill_valve;
};
```

Testable hardware interactions

```
using hw_register = std::uint32_t volatile;

struct gpio_memory_layout {
  hw_register mode;                   // MODER
  hw_register output_type;            // OTYPER
  hw_register output_speed;           // OSPEEDR
  hw_register pull_up_down_register;  // PUPDR
  hw_register input_data;             // IDR
  hw_register output_data;            // ODR
  hw_register bit_set_reset;          // BSRR
  hw_register locked;                 // LCKR
  hw_register alternate_function_low;  // AFR (low word)
  hw_register alternate_function_high; // AFR (high word)
};
```

Testable hardware interactions

```
class port {
public:
  void configure(input const &configuration);
  void configure(output const &configuration);

  // Much detail omitted ...

  pin_state read(uint16_t pin) const;
  void write(uint16_t pin, pin_state state);
  void toggle_pin(uint16_t pin);

private:
  gpio_memory_layout memory;
};
```

Testable hardware interactions

```
TEST_CASE("The GPIO port") {

  stm32::gpio_memory_layout gpio_memory{.bit_set_reset = 0x0};
  auto gpio_port = new (&gpio_memory) stm32::port;

  using register_bits = std::bitset<32u>;

  SECTION("When pin set low, sets bit in lower BSRR half-word") {
    gpio_port->write(pin_number::pin_3, pin_state::reset);

    REQUIRE(register_bits(gpio_memory.bit_set_reset).test(3u));
  }

  SECTION("When pin set high, sets bit in upper BSRR half-word") {
    gpio_port->write(pin_number::pin_3, pin_state::set);

    REQUIRE(register_bits(gpio_memory.bit_set_reset).test(19u));
  }
}
```

Testable RTOS interactions

- RTOS tasks tend to accrete a lot of responsibility in RTOS apps
- Really, too much : often a sign of insufficient separation
- ...but they're a core component in event-driven RTOS systems
- The `while(1){}` idiom is a barrier to meaningful TDD
- We want to make our tasks a testable, encapsulated unit

```
typedef void (*TaskFunction_t)( void * );

BaseType_t xTaskCreate(TaskFunction_t pvTaskCode,
                       const char *pcName,
                       const uint16_t usStackDepth,
                       void *pvParameters,
                       UBaseType_t uxPriority,
                       TaskHandle_t *const pxCreatedTask);
```

A testable blinky task

```
template <typename T = rtos::forever>
class idle_task {
public:
  explicit idle_task(stm32::port &gpio) : gpio_port{gpio} {}

  void operator()() {

    T loop;

    while (loop()) {
      gpio_port.toggle_pin(LD2_Pin);
      vTaskDelay(delay_ticks);
    }
  }

private:
  stm32::port &gpio_port;
};
```

A testable task...actually tested

```
TEST_CASE("The idle indicator task") {

  stm32::gpio_memory_layout gpio_memory{.output_data = 0x0};
  auto gpio_port = new (&gpio_memory) stm32::port;

  SECTION("Run a single time, toggles the LED to on from off") {
    platform::idle_task<rtos::once> blinky_task{*gpio_port};

    blinky_task();

    REQUIRE((gpio_memory.output_data & LD2_Pin) != 0);
  }

  SECTION("Run twice, toggles the LED back to its original state") {
    platform::idle_task<rtos::twice> blinky_task{*gpio_port};

    blinky_task();

    REQUIRE((gpio_memory.output_data & LD2_Pin) == 0);
  }
}
```

Enabling testable tasks : loop policy

```
struct forever {
  bool operator()() { return true; }
};

template <std::size_t T> struct times {
  std::size_t run_count = T;
  bool operator()() { return (run_count-- != 0); }
};

using once = times<1u>;
using twice = times<2u>;
```

Enabling testable tasks : scheduling

```
template <typename T_task>
void rtos_task(void *task_object) {
  (*static_cast<T_task *>(task_object))();
}


template <typename T>
void create_task(T *task_object, const char *const name,
                 uint16_t depth, unsigned long priority) {

  xTaskCreate(rtos_task<T>, name, depth, task_object,
              priority, nullptr);
}
```

Real-deal use of the testable task

```
// Create the blinky task, injecting the GPIO port

static idle_task health_indicator{gpio_a};



// Before starting the RTOS scheduler...

rtos::create_task(
  &health_indicator, "Blink", 128u, 0ul);
```

Testability with 3rd party libraries

Testability with 3rd party libraries

STUB

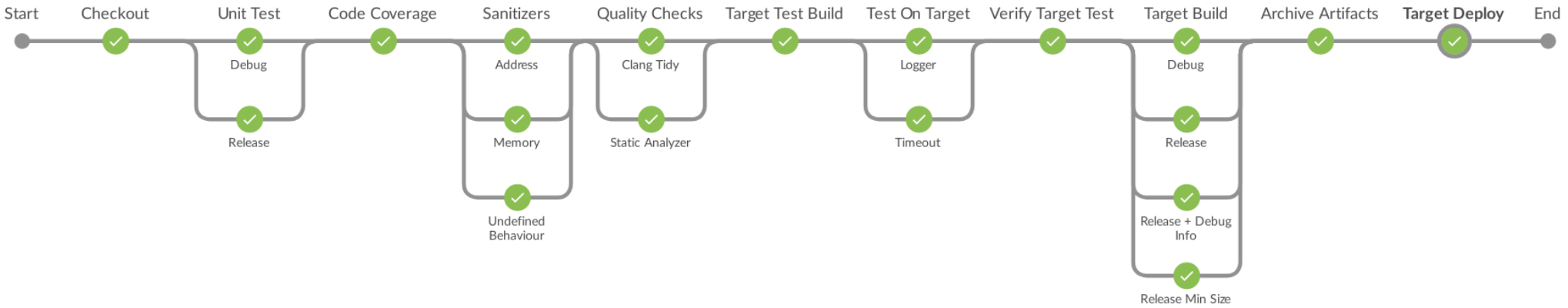
Testability with 3rd party libraries

- Yay for linkers!
- Stub as a default – and hardcode returns as far as possible
- Test Spy when you need to know how the library was called
- Stub only what you need for test – usually a fraction of a lib
- Resist temptation to mock – it's almost always the wrong tool
- Consider using the Fake Function Framework as a default:

https://github.com/meekrosoft/fff

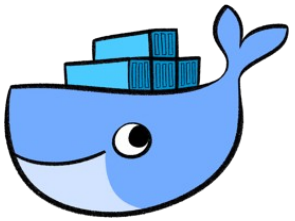
The CI build you owe to yourself

Embiggen your build



Let the continuous integration build **amplify** the tests.

Working to make your build build



Docker provides a Jenkins server, cross-compilation toolchains and support tools from a Dockerfile.

Server picks up Jenkinsfile from Git repository, build and runs pipeline on change.

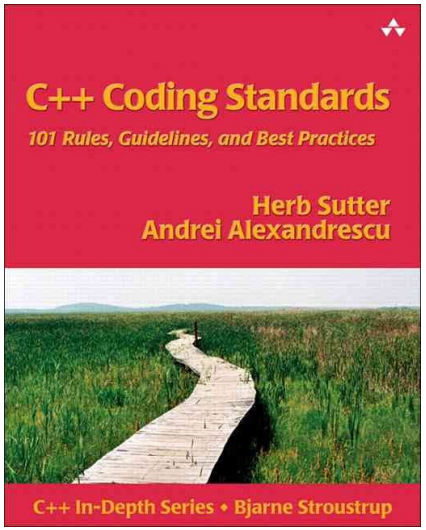
Toolchain definition files, standardised target names for Jenkins to run, detailed knowledge on build and link.

Repository for your sources and colocated Jenkinsfile. Commit hooks to initiate build.

Always remember:

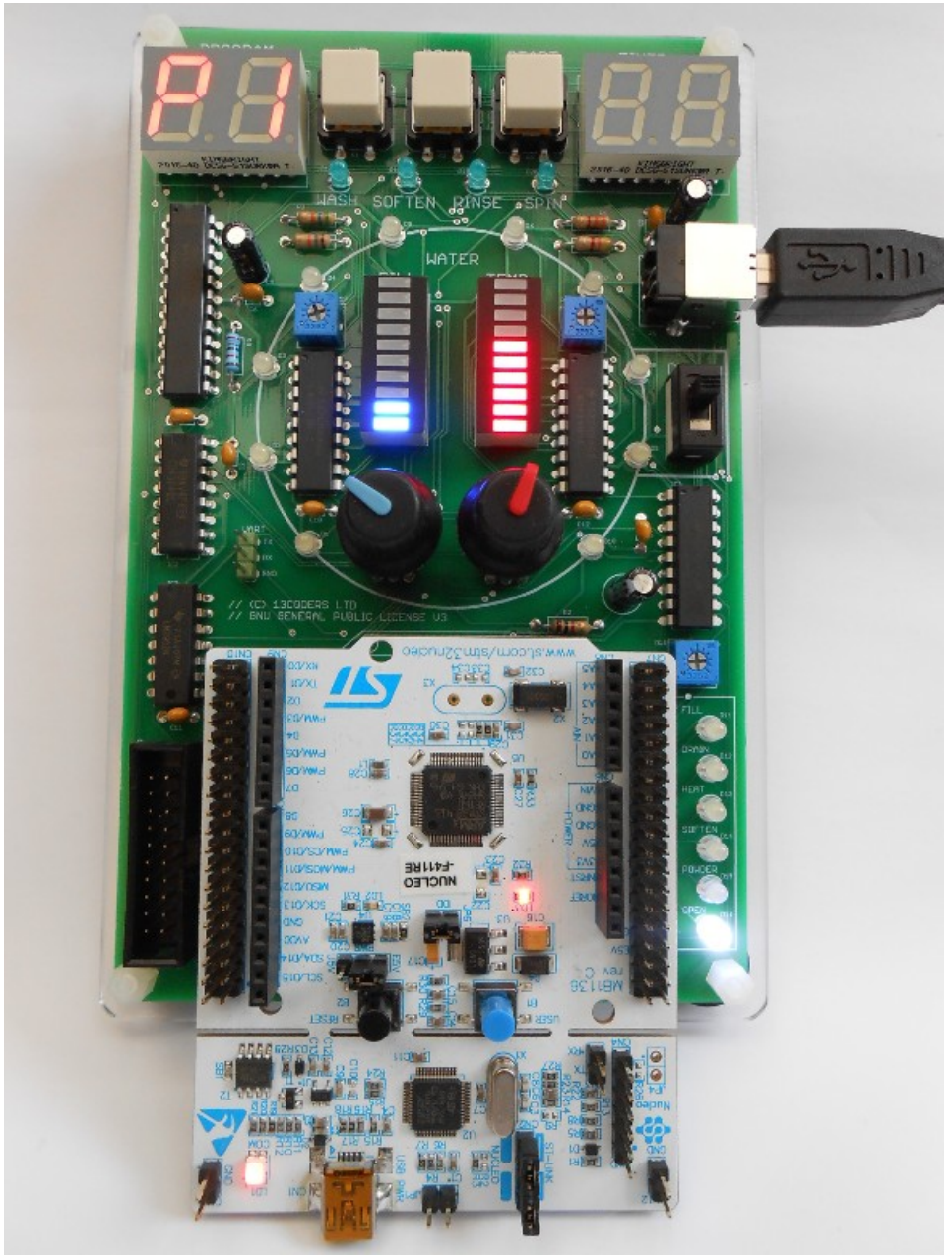
It is far, far easier to make a correct program fast than it is to make a fast program correct.

So, by default, don't focus on making code fast; focus first on making code as clear and readable as possible (see Item 6). Clear code is easier to write correctly, easier to understand, easier to refactor—and easier to optimize. Complications, including optimizations, can always be introduced later—and only if necessary.



"It is far, far easier to make a correct program fast than it is to make a fast program correct."

Meet the washing machine



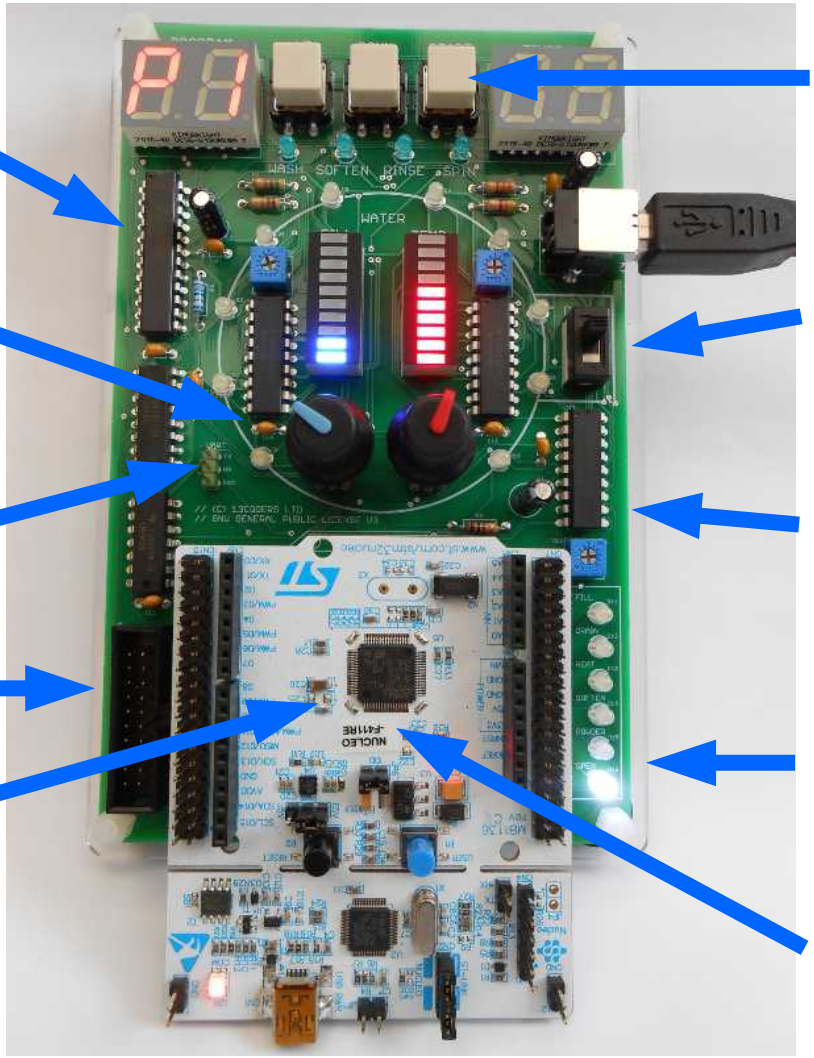
WAT

It's a platform for "deliberate practice" on embedded systems - practice of design, TDD and continuous integration.

It's a hardware simulation of a washing machine, design is released under GPL v3.0.

And it's trickier than you might imagine...

Washing machines are suprisingly challenging



SPI serial LED driver

2 x analog "sensors"

UART communication

20-pin JTAG header

Choice of ARM Cortex-M4 or M0 boards

Multiple discrete GPIO

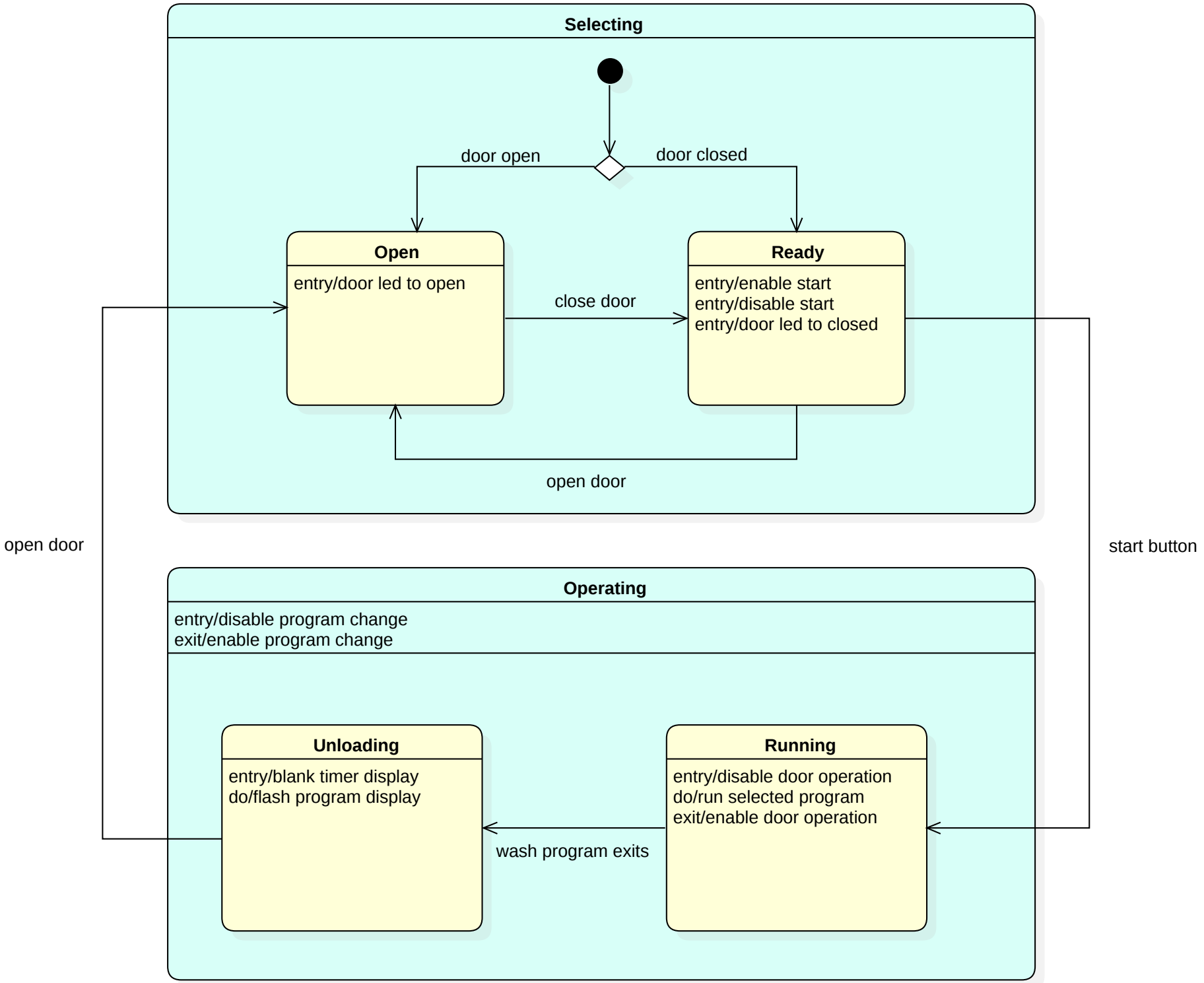
Latching switches and momentary switches

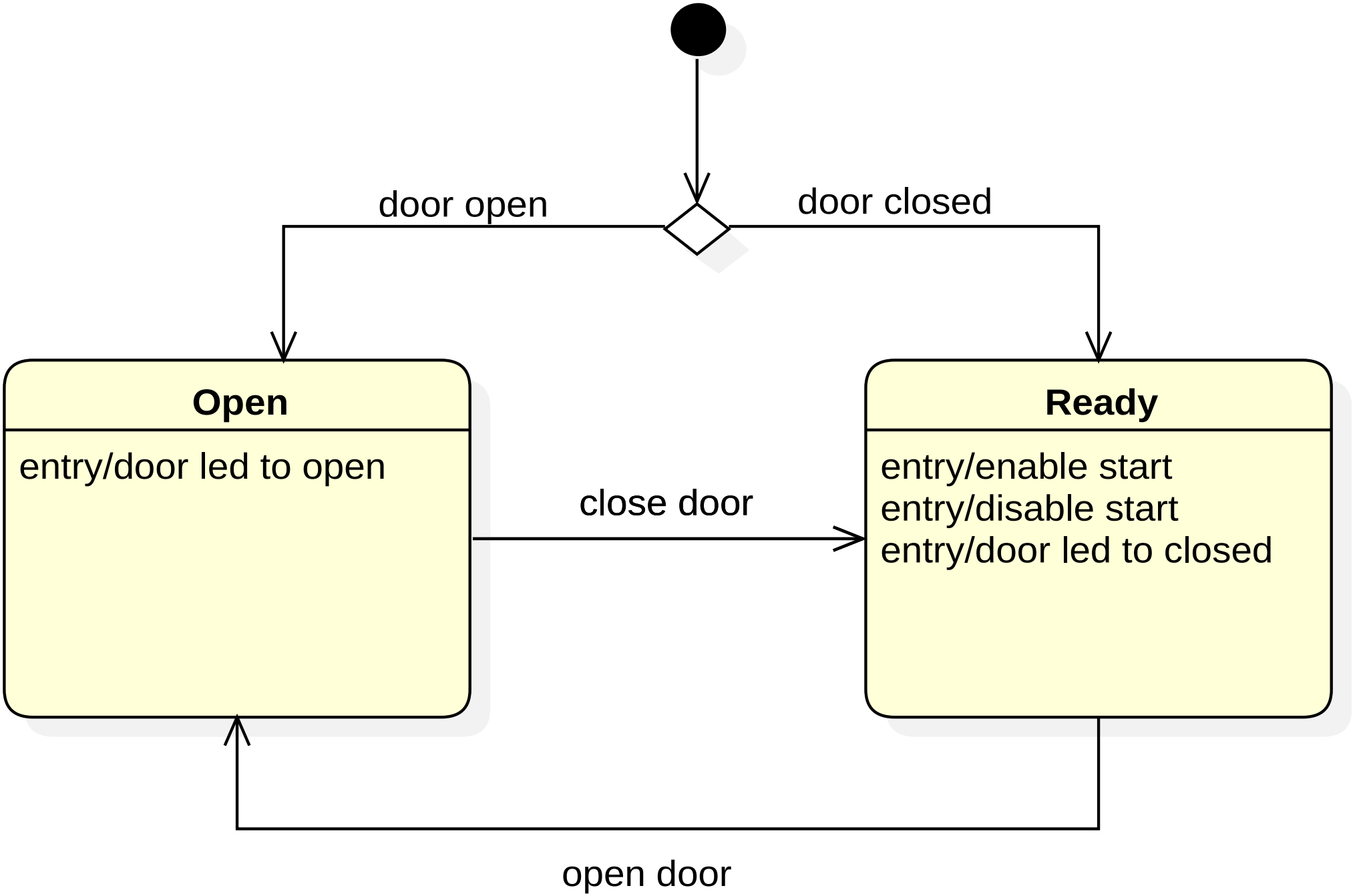
PWM "motor"output (+ low-pass filter)

Many, many, many LEDs

Timers for cycle countdown

It’s demo time...


Selecting
door open
door closed
Open
entry/door led to open
close door
Ready
entry/enable start
entry/disable start
entry/door led to closed
open door
open door
start button
Operating
entry/disable program change
exit/enable program change
Unloading
entry/blank timer display
do/flash program display
wash program exits
Running
entry/disable door operation
do/run selected program
exit/enable door operation


door open
door closed
Open
entry/door led to open
close door
Ready
entry/enable start
entry/disable start
entry/door led to closed
open door

Wrapping up

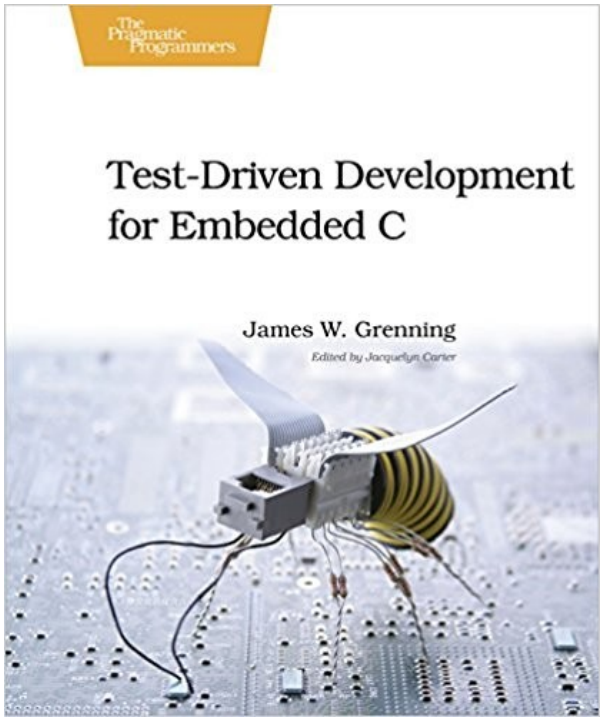
The important parts

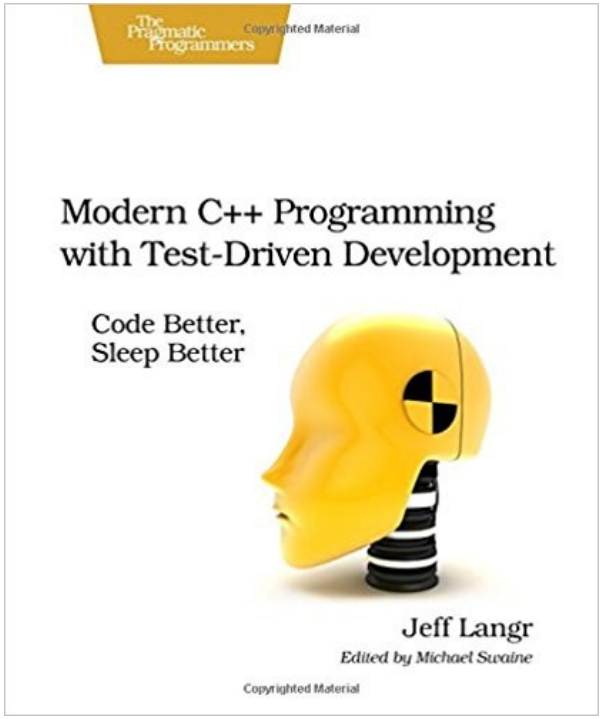
- Small increments mean containable mistakes
- Testability needs to be a factor in your design
- Adapt libs to fit your design when needed
- If it's not impeding you, stop worrying about it
- The HAL will likely be your testability barrier
- Optimise for rapid development on the host

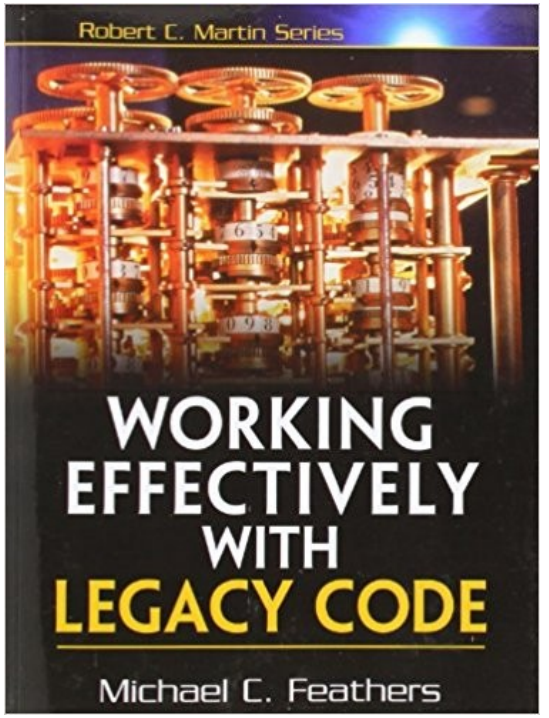
The important parts

- Tiny devices, but a DevOps mindset
- Automation frees time for high-value activity:
  - Use specialised tools for the target
  - Profile / trace / ‘scope / logic analyse
  - These are for insight, not to test for functional correctness
- Test both on and off the target continuously

Your Reading List







That's it !

Thanks for listening

mike@13coders.com
www.13coders.com

Questions ?