Embedded Systems Best Practices

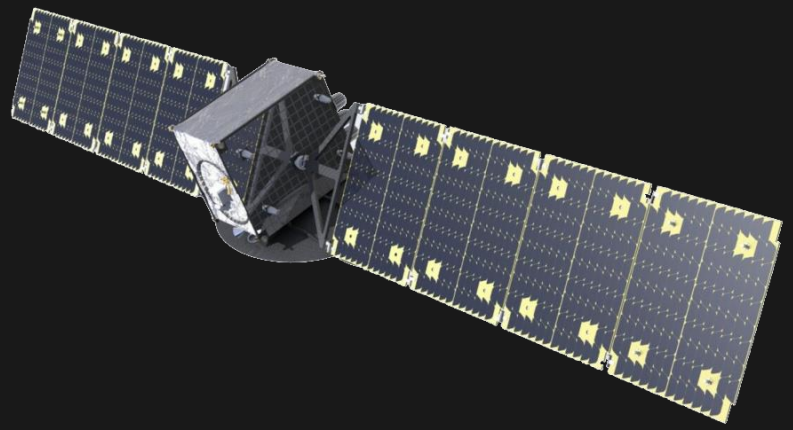
Musings From the Aero Industry

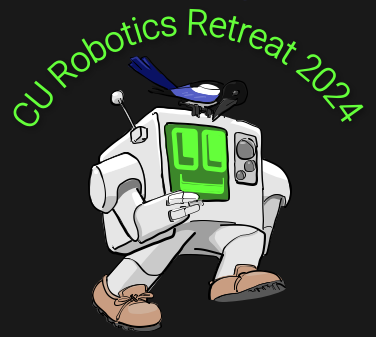
Ryan Draves

11/23/2024

About Me

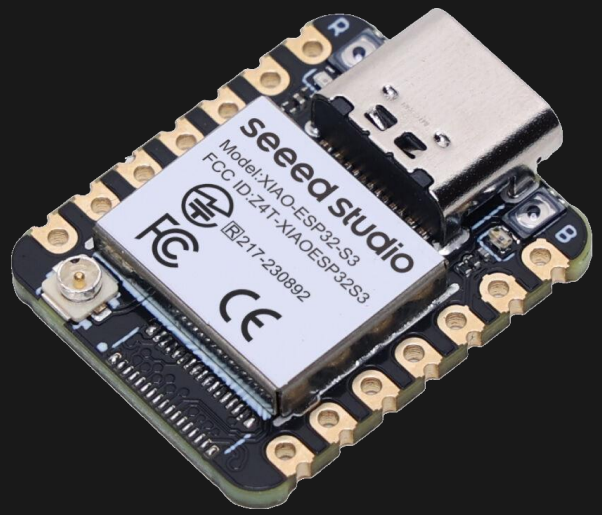
- Ryan Draves
- 1st year Ph.D. student
- Background is CS, Robotics, & Aero
- Spent 3 years at Astranis
  - Hardware test infrastructure
  - Embedded systems development
  - Dev-ops enthusiast
  - Bazel merchant





About This Presentation

- Some best-practices and design philosophies
- A small project with some ESP32 modules & ultrasonic sensors
- A bonus challenge



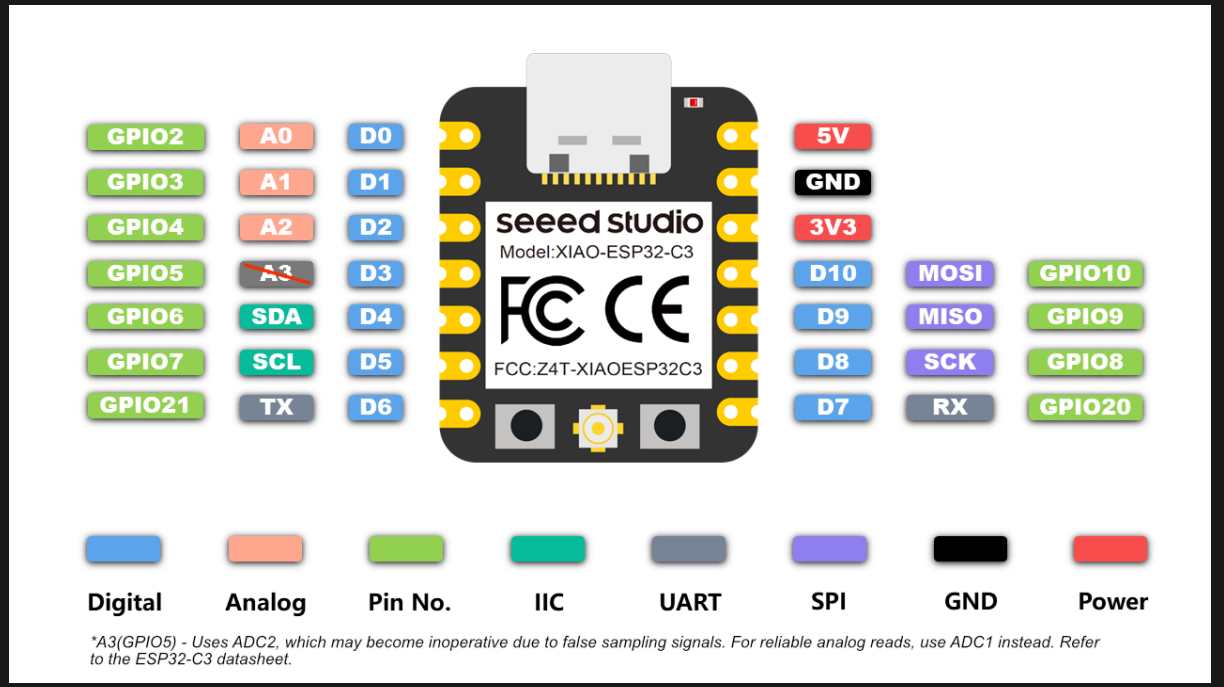
Project Details

- Start installing now if you haven't set up the environment yet
- Install steps:
  - Clone RyanDraves/robo-24-workshop & follow the README.md
  - Open main/main.cc, read the comments at the top
  - These steps will be in the corner of the following slides

Project Details

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

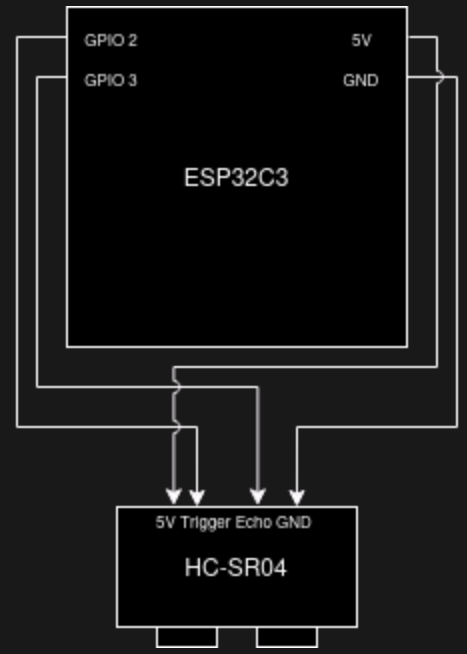
- The XIAO ESP32-C3 is a neat microcontroller
  - RISC-V CPU with floating point support
  - WiFi & Bluetooth modules
  - Lithium battery management
  - 400KB SRAM & 4MB flash
  - Assorted cryptography support
  - Direct USB support
  - Comprehensive SDK
    - Includes most of stdlib

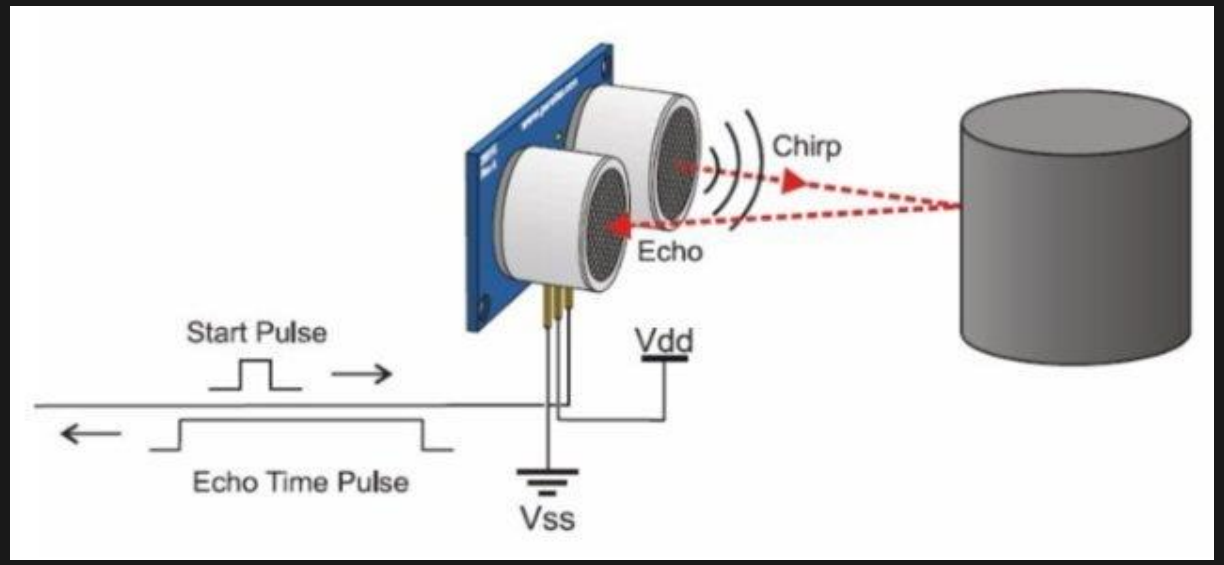


Project Details

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

- The HC-SR04 is an ultrasonic distance sensor
  - In theory, you can poll the sensor as soon as the previous measurement returns
  - In practice, this is a bad idea, and the datasheet recommends a minimum time between measurements





Project Details

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

- The repo contains a naïve driver and request handler to work from
- A "bad" client will ignore the datasheet's timing requirement
- The challenge is to re-implement the request handler so it can, on the *first* test, handle bad clients
- Bonus: improve the HITL test for the device
- Bonus: Develop a script to empirically find the fastest the device can be polled

Project Details

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

Example Demo

Project Tips

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

- stdout is wired to the shell console; feel freet to print debug with std::cout or printf
- The code comments link to useful references for ESP32 platform & HC-SR04 specific details

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

Best Practices

Misc. topics ~related to the project,
Mildly C++-oriented

Sanitize Your Inputs

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

- Firmware should not assume well-formed inputs
- Motivation is to isolate failures
- Hardware inputs can also fail or have unexpected values
  - Does the firmware depend on nominal values to operate safely?
- Generically this is known as defensive programming

Sanitize Your Inputs

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

- Malformed requests are a common failure case
- Examples:
  - Programmer error (made a bad client)
  - User error (client allowed sending bad requests)
  - Version incompatibilities make (formerly) good requests nonsensical
  - Failure propagation
    - E.g. a higher-level system (say a robot's central compute) is bugged and sending dangerous commands to an embedded subsystem

Sanitize Your Inputs

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

- HC-SR04 example: averaged samples
- ```
  ...
  class MesaurementRequest(TypedDict):
      num_samples: int  # Hope no one picks (-inf, 0)...
  ```
- What if we request 0 samples?


  ```

Sanitize Your Inputs

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

- HC-SR04 example: averaged samples
-
```python
...
class MesaurementRequest(TypedDict):
    num_samples: int  # Hope no one picks (-inf, 0)...
```

- What if we request 0 samples?

```
In [1]: client.request_measurement()
Out[1]: {'distance_mm': 4294967295, 'timestamp_ms': 9011}
```

Optional values are safe defaults

RyanDraves/robo-24-workshop (Github)
README.md, main/main.cc

- Background: a default is any value a variable takes on when one isn't explicitly provided
- Examples:
  - Default arguments in functions
  - 0 is the default for integers
  - Undefined variables *may* default to previous values in memory
    - "Undefined behavior"

```cpp
// Some safety wheels were taken off the compiler for this
uint32_t x;
uint32_t *y = new uint32_t;
std::cout << "Mysterious stack variable: " << x << std::endl;
std::cout << "Mysterious heap variable: " << *y << std::endl;
delete y;
```

Optional values are safe defaults

RyanDraves/robo-24-workshop (Github)
README.md, main/main.cc

- Default values (in inputs or outputs) can create ambiguity and dangerous assumptions
  - E.g. ambiguity between 0 and “not there”
- `optional` classes (from the STL or your favorite library) clearly signal when a value is present
  - E.g. if you receive a request to read from flash, but the address to read is not set, you know the request is ill-formed and not a request to read address 0

```
// Some safety wheels were taken off the compiler for this
uint32_t x;
uint32_t *y = new uint32_t;
std::cout << "Mysterious stack variable: " << x << std::endl;
std::cout << "Mysterious heap variable: " << *y << std::endl;
delete y;
```

```
In [4]: client.request_measurement()
2024-11-18 22:48:59,632 - INFO - Mysterious stack variable: 0
2024-11-18 22:48:59,632 - INFO - Mysterious heap variable: 1070131828
```

Optional values are safe defaults

RyanDraves/robo-24-workshop (Github)
README.md, main/main.cc

- Simple example: request handling with std::optional
  - Now we know if num_samples was provided

```
...
struct MeasurementRequest {
    std::optional<uint8_t> num_samples;
};
```

```
void handle_measurement_request(const MeasurementRequest& req, HcSr04& sensor, Measurement *meas) {
    if (!req.num_samples.has_value()) {
        // Hmm, that's weird. Better not do anything.
        return;
    }

    for (uint8_t i = 0; i < req.num_samples.value(); i++) {
```

Request Handling Methodologies

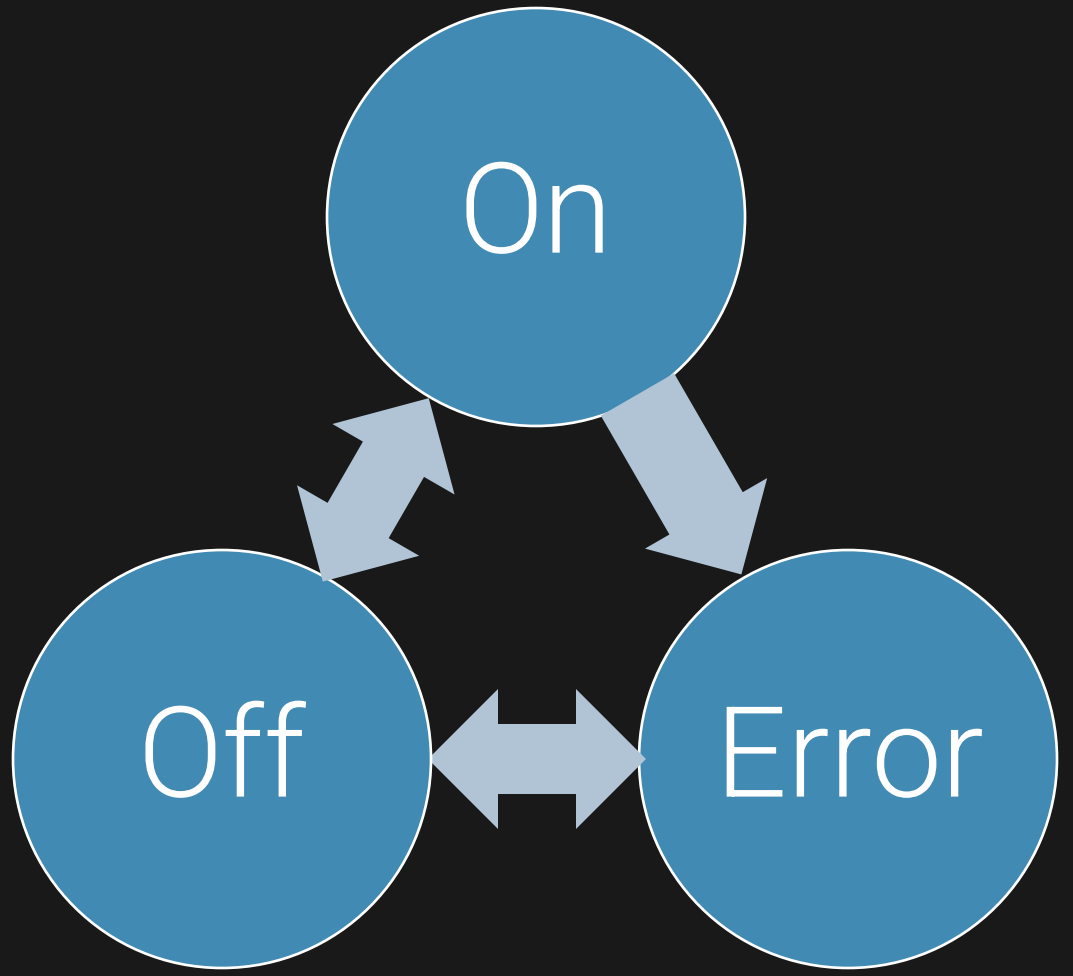
RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

- Naïve: do what the client asks
- Strict: raise* an error upon bad requests
- Engineer: push the problem somewhere else (return most recent value)
- Tri-state: make the problem a state machine

Tri-State Logic

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

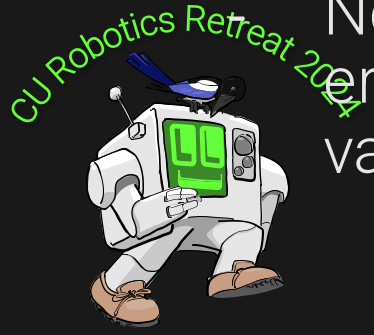
- Simple state machine:
  On, Off, Error
- Extendable to *N* states + Off + Error
- Any state can go to an error state; an error state can only go an “off” state

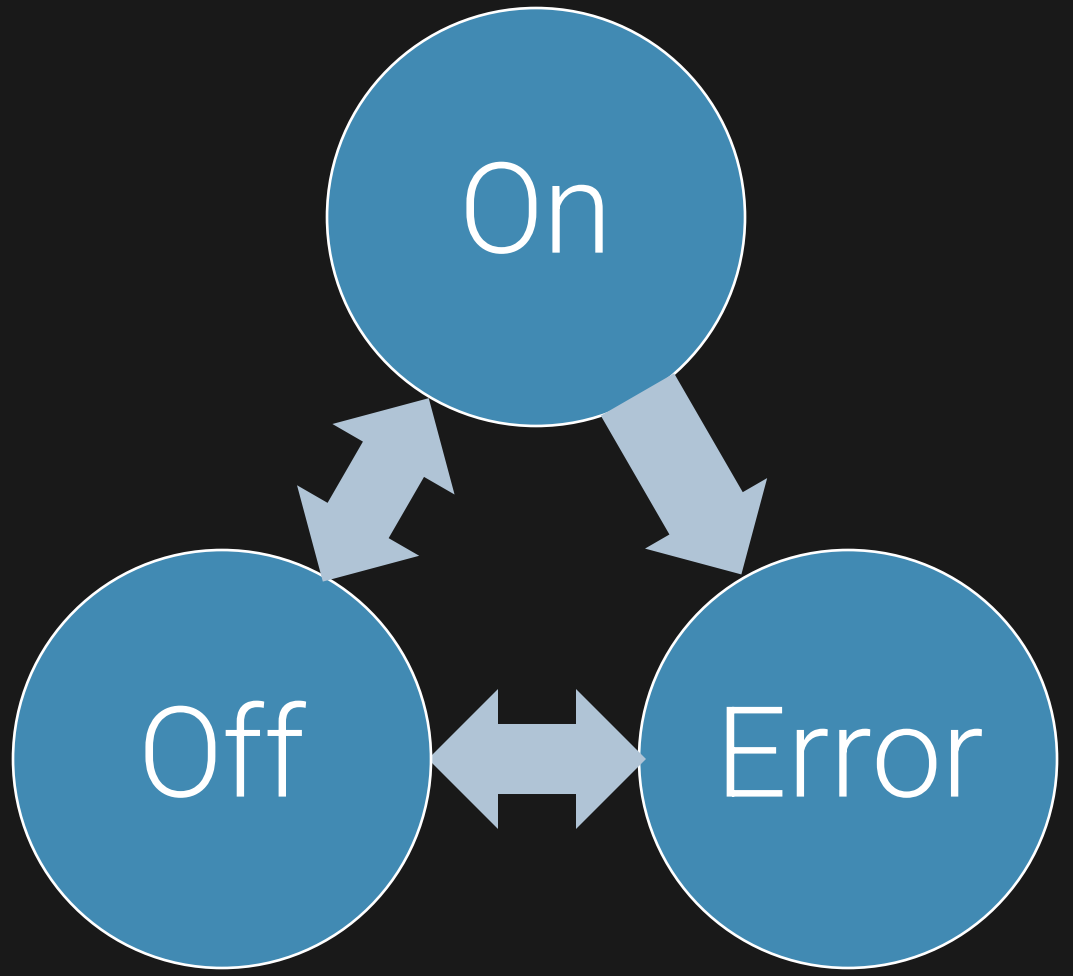


Tri-State Logic

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

- The error state requires *acknowledgement*
- Imagine other software trying to turn on a faulty system
  - E.g. stuck in a naïve “if not on, turn on” loop
- The tri-state breaks the loop and prevents further errors
  - Neat trick: in an enum, make the error state the uninitialized value (0)

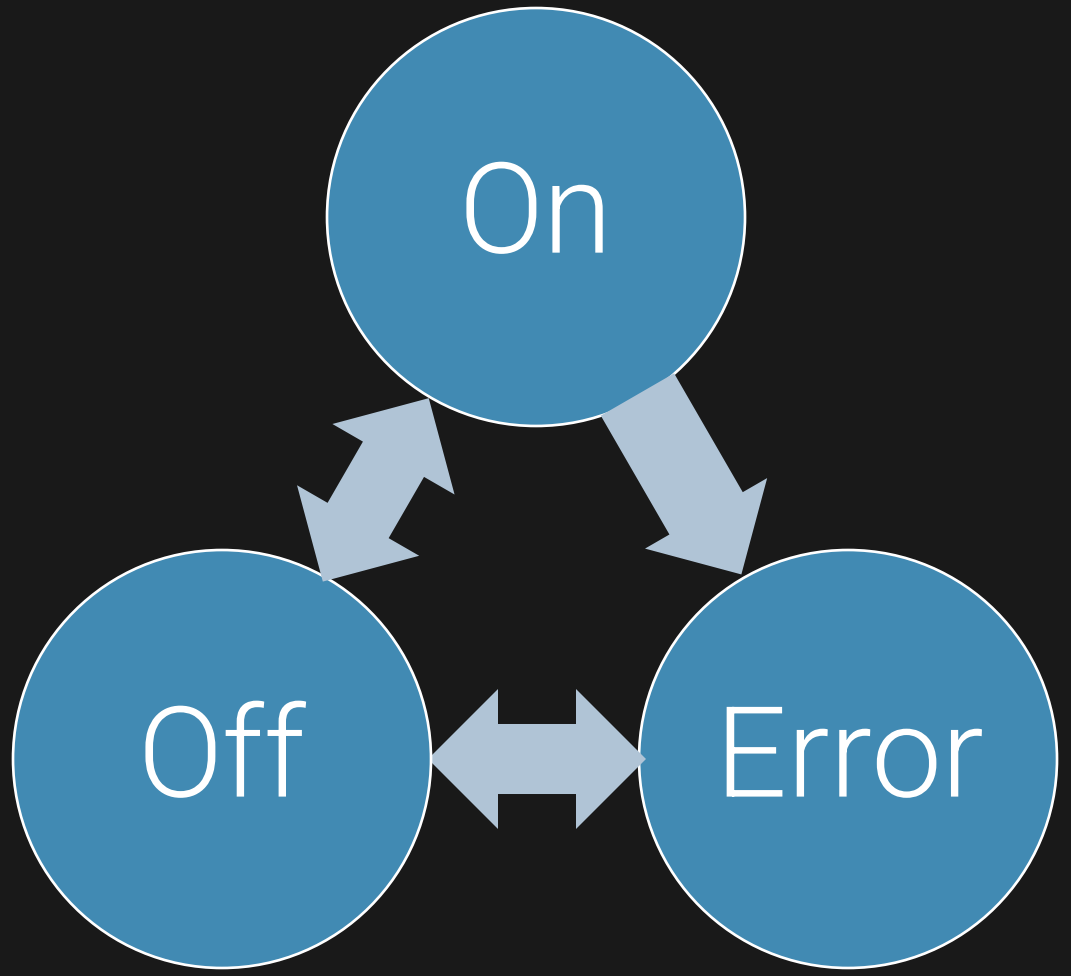




Tri-State Logic

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

- Application: if our sensor rate limit was a hardware risk, we might want to enter an error state upon a bad request
- Cons: there's mental overhead for an operator to deal with error states
  - What's the error?
  - Is it safe to reset the error?
  - How often do I have to reset it?



Time is a Function Argument

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

- Embedded systems frequently deal with timing requirements
- Testing time-dependent can be tricky, tedious, and unreliable
- Avoid reading and using a timestamp in the same function
    - Pass the timestamp as a function argument instead


```cpp
uint32_t g_last_time = 0;
constexpr uint32_t g_period_ms = 1000;

uint32_t get_time_ms() {
    // Read the clock (usually ms since boot)
    return 0;
}

void my_function() {
    // Get the current timestamp
    uint32_t timestamp = get_time_ms();

    // Compare to our last timestamp
    if (timestamp - g_last_time < g_period_ms) {
        // Don't do the thing
        return;
    }

    // Update the last timestamp
    g_last_time = timestamp;

    // Do the thing
}
```

Time is a Function Argument

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

- Embedded systems frequently deal with timing requirements
- Testing time-dependent can be tricky, tedious, and unreliable
- Avoid reading and using a timestamp in the same function
    - Pass the timestamp as a function argument instead

```cpp
uint32_t g_last_time = 0;
constexpr uint32_t g_period_ms = 1000;

uint32_t get_time_ms() {
    // Read the clock (usually ms since boot)
    return 0;
}

void my_timed_function(uint32_t timestamp) {
    // Compare to our last timestamp
    if (timestamp - g_last_time < g_period_ms) {
        // Don't do the thing
        return;
    }

    // Update the last timestamp
    g_last_time = timestamp;

    // Do the thing
}
```

Time is a Function Argument

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

Pros:

- *Greatly* simplifies logic testing
- Lets you test long-duration cases otherwise unsuitable for unit tests
- Eliminates test flakiness; you know what the timestamp will be, because you set it

```
void test_my_function() {

    my_function();
    ASSERT(some initial behavior);

    sleep_ms(g_period_ms - 1);
    my_function();
    ASSERT(some behavior);

    sleep_ms(1);
    my_function();
    ASSERT(some other behavior);

    sleep_ms(-1);  // Oops, not valid
    my_function();
    ASSERT(guess we can only test monotonic clocks);

    sleep_ms(g_period_ms * 999);
    my_function();
    ASSERT(anyone want coffee?);
}
```

Time is a Function Argument

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

Pros:

- *Greatly* simplifies logic testing
- Lets you test long-duration cases otherwise unsuitable for unit tests
- Eliminates test flakiness; you know what the timestamp will be, because you set it

```
void test_my_timed_function() {
    uint32_t timestamp = 0;
    my_timed_function(timestamp);
    ASSERT(some initial behavior);

    timestamp += g_period_ms - 1;
    my_timed_function(timestamp);
    ASSERT(some behavior);

    timestamp += 1;
    my_timed_function(timestamp);
    ASSERT(some other behavior);

    timestamp -= 1;
    my_timed_function(timestamp);
    ASSERT(behavior when the clock is not monotonic);

    timestamp += g_period_ms * 999;
    my_timed_function(timestamp);
    ASSERT(some other behavior);
}
```

Time is a Function Argument

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

Pros:

- *Greatly* simplifies logic testing
- Lets you test long-duration cases otherwise unsuitable for unit tests
- Eliminates test flakiness; you know what the timestamp will be, because you set it

Cons:

- Execution time in the method before the timestamp is used needs to be trivial
- Passing an extra function arg everywhere can be tedious

Time is a Function Argument

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

Application:

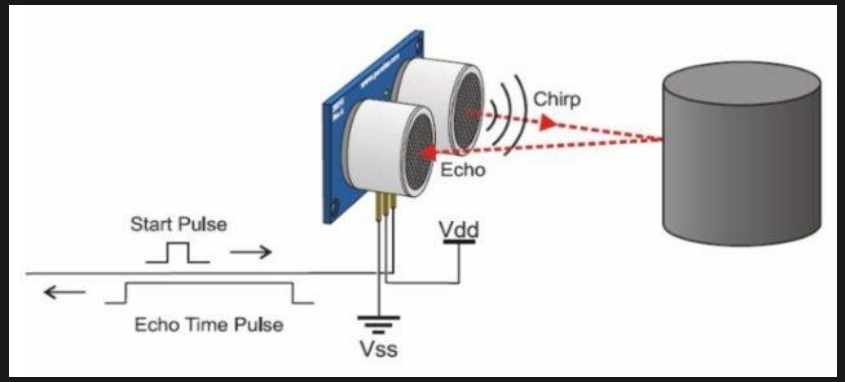
- The `handle_measurement_request` method in the project can be tested this way
  - Just need to mock out the hardware interactions in the sensor

```
void handle_measurement_request(uint64_t timestamp_us, HcSr04& sensor, Measurement *meas) {
    meas->timestamp_ms = timestamp_us / 1000;
```

Fractions Without Floats

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

- On some systems, floating point numbers may not be available
- On most systems, integer operations are meaningfully faster
- Trick: use unreduced fractions
- Example: Calculating distance (in millimeters) from pulse width (in microseconds) in the sensor driver
  - Need to divide by 5.8, same as multiplying by 1000 / 5800




```
// Calculate the distance
uint32_t pulse_width_us = echo_end_us - echo_start_us;
// Calculated from the assumed speed of sound in air at sea level (~340
// m/s); pulse. Constant provided in the datasheet.
uint32_t distance_mm = pulse_width_us * 1000 / 5800;
```

Fractions Without Floats

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

- Be careful about overflows
  - In this example:

```
In [13]: 2**32 / 1000000 / 60 / 60
Out[13]: 1.193046471111111

In [14]: 2**32 / 5800
Out[14]: 740511.6027586206
```

- Neat trick if we would've overflowed: cast to `uint64_t` in the first operation

```
// Calculate the distance
uint32_t pulse_width_us = echo_end_us - echo_start_us;
// Calculated from the assumed speed of sound in air at sea level (~340
// m/s); pulse. Constant provided in the datasheet.
uint32_t distance_mm = (uint64_t)(pulse_width_us * 1000) / 5800;
```

Static Initialization Order Fiasco

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

- A real concept with a real cppreference page
- In C++, the compiler “arbitrarily” picks the initialization order of static objects
- Non-trivial constructors in firmware can incur undefined behavior
  - E.g.: IO register configuration occurs at or shortly after global variable initialization, but a class setting default pin states attempts to do so during global variable initialization

Static Initialization Order Fiasco

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

- Mitigations
  - Construct on First Use idiom (chicanery to force initialization order)
    - Much like the `property` deferred initialization pattern in Python

```python
@property
def port(self) -> str:
    if self._port is None:
        self._port = self._find_esp32s3()
    return self._port
```

  - Only write trivial (no pin interaction) constructors
  - Move pin initialization to a non-constructor method (like `init`)

Measuring Stack Usage

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

- Running out of stack is fairly common
- *Undefined behavior* and general crashing are commonplace
- Useful to measure how much stack is being used
- FreeRTOS (& similar) can read the “high watermark”
  - Usually by placing a default value like 0xAA (0b10101010) in memory and measuring the last non-default memory location in the stack
- Lets you debug stack utilization as well as limit stack size (when using statically allocated task stacks)


```
2024-11-18 23:51:35,555 - INFO - Stack watermark: 2228 / 3584 bytes remaining
```

Testing Methodologies

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

- Unit testing
- Integration testing
- Hardware integration testing

Testing Methodologies

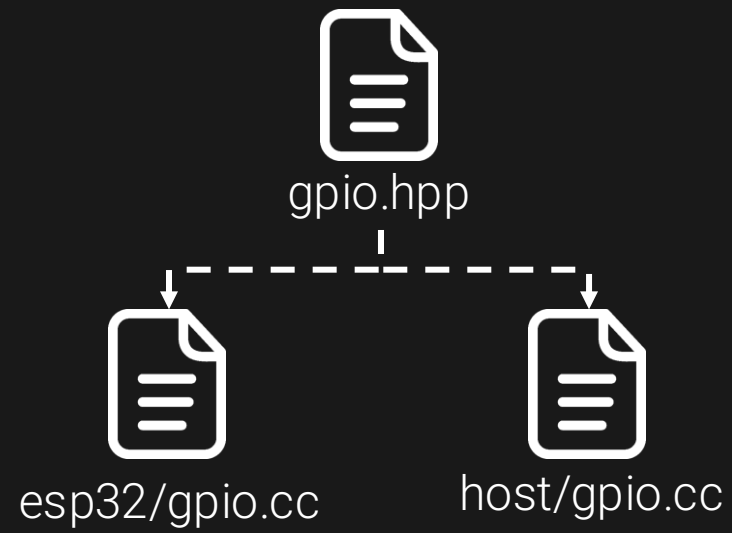
RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

- Unit testing
- Integration testing
- Hardware integration testing

Testing Methodologies

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

- Low-level code *can* be hard to unit test
- It can also be easy
- Examples:
  - Time is a function argument
  - Platform-based dependency selection
  - Separation of concerns



Testing Methodologies

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

- Examples for this project:
  - `handle_measurement_request`
  - `HcSr04` driver

```cpp
uint32_t HcSr04::get_distance_mm() {
    // Trigger the sensor
    gpio_set_level(trigger_, 1);
    delay_microseconds(10);
    gpio_set_level(trigger_, 0);

    // Wait for the echo to go high for up to kMaxDistanceUs
    uint32_t timeout_us = esp_timer_get_time() + kMaxDistanceUs;
    while (!gpio_get_level(echo_)) {
        if (esp_timer_get_time() > timeout_us) {
            return kMaxDistanceMm;
        }
    }
```

Testing Methodologies

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

- Unit testing
- **Integration testing**
- Hardware integration testing

Testing Methodologies

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

- Simulation
  - Host compiled firmware
  - Forwards IO interactions to a simulation of the hardware
- Emulation
  - Target compiled firmware run on an emulator
  - May also forward IO interactions to a simulator
- Both enable client <-> firmware integration testing

Testing Methodologies

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

- Unit testing
- Integration testing
- Hardware integration testing

Testing Methodologies

RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

- Full testbed connected to hardware
- Manual & automated integration testing
- *Cacheable* with the right build infrastructure
  - E.g. every code change runs hardware regression tests, only relevant (client / firmware) changes *actually* run the tests

Testing Methodologies

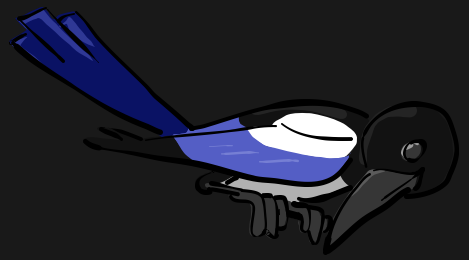
RyanDraves/robo-24-workshop
(Github)
README.md, main/main.cc

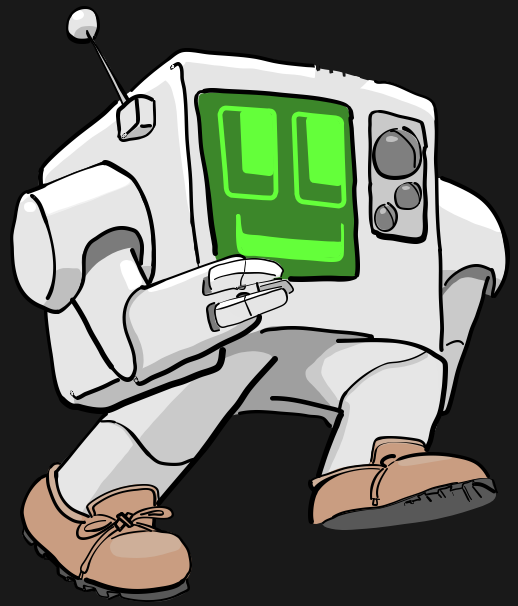
- Examples for this project
  - A simple HITL test is included that checks the measurement rate against the datasheet's suggestion

```
dravesr@dravesr-recuv:~/src/robo-24-workshop$ pytest pytest_dut.py
============================================================================= test session starts
platform linux -- Python 3.12.3, pytest-8.3.3, pluggy-1.5.0
rootdir: /home/dravesr/src/robo-24-workshop
plugins: ignore-test-results-0.2.2, rerunfailures-14.0, embedded-1.12.0, timeout-2.3.1
collected 2 items

pytest_dut.py .F

================================================================================== FAILURES =====
```






CU Robotics Retreat 2024


CU Robotics Retreat 2024