To the Stratosphere and Beyond
Rust at 100,000ft

Konstantinos Stathopulos

About Myself

ACC
ARAPAHOE COMMUNITY COLLEGE

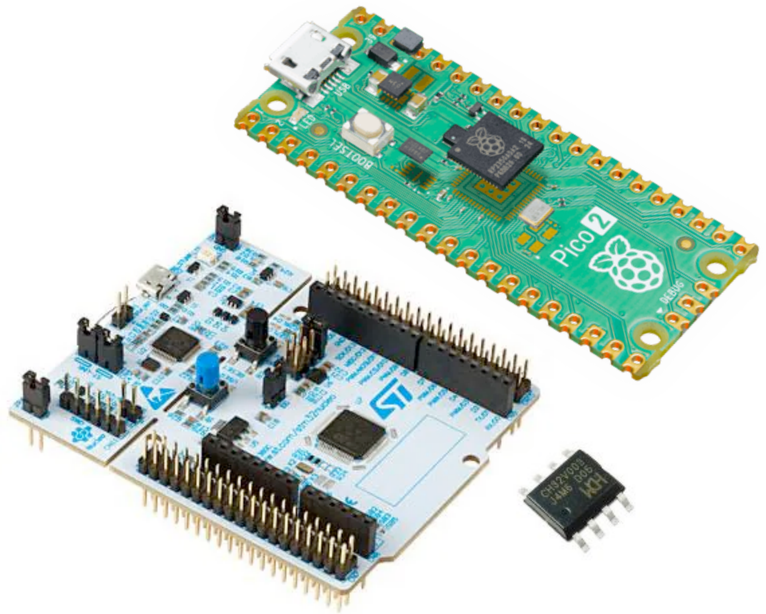
COSGC
COLORADO SPACE GRANT CONSORTIUM

COLORADO SPACE GRANT
DEMOSAT
EDGE OF SPACE SCIENCES

A word about microcontrollers

Stepping up to 32-bit

- Traditionally Arduino
  - WB8ELK Bill Brown’s talks at GPSL
- AVR *is* possible, but very different
- Opens up a world of possibilities
- Why you might want to stick with Arduino
  - The code is not (usually) the point
- When to consider going 32-bit?



What is a High Altitude Balloon?

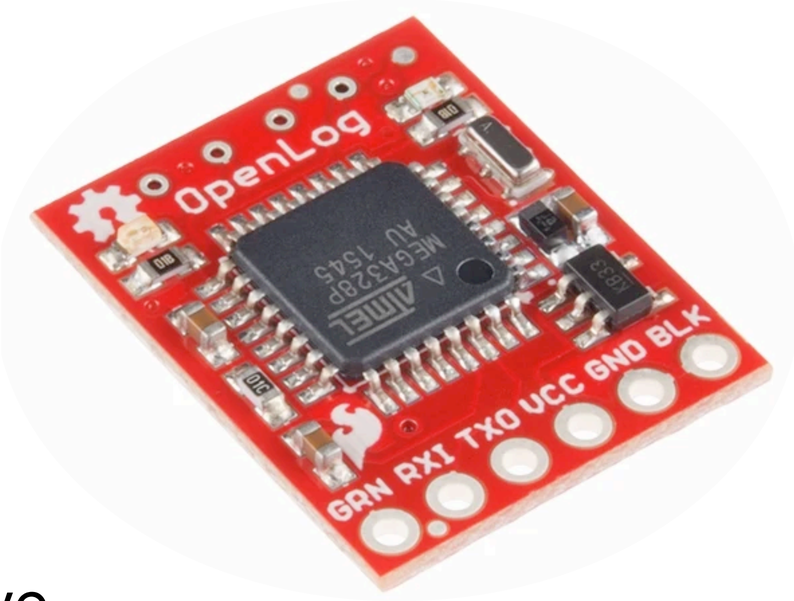
and how high does it go?

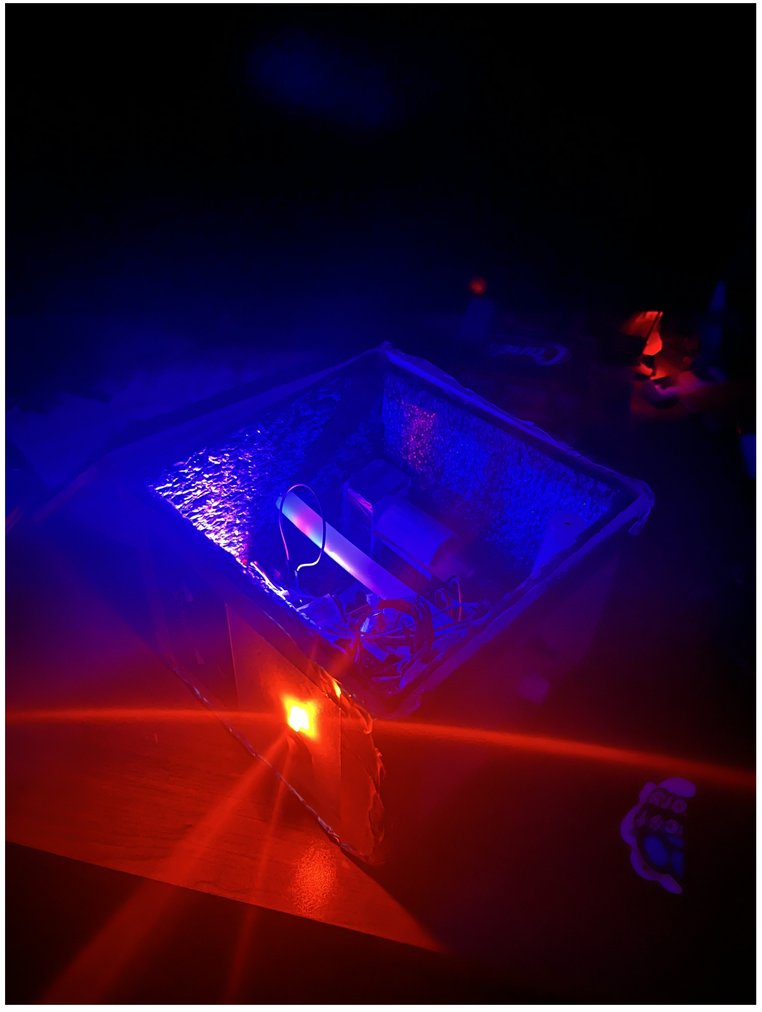
- Traditional Heavy Balloons
  - ~100,000ft
  - Thousands of grams
- Pico Balloons
  - ~40,000ft
  - Dozens of grams
  - All the way around the globe

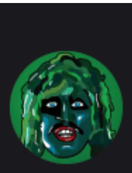
Data Logging

and why it pays to be the dumbest guy in the room

- SparkFun OpenLog
- One wire, records everything it sees over serial
- Lets our payload forget about saving data
- WYSIWYG over Serial Monitor
- `println!` and `Serial.log()`
- Only works because of where and when we retrieve
  - RockSat SD Card horror stories

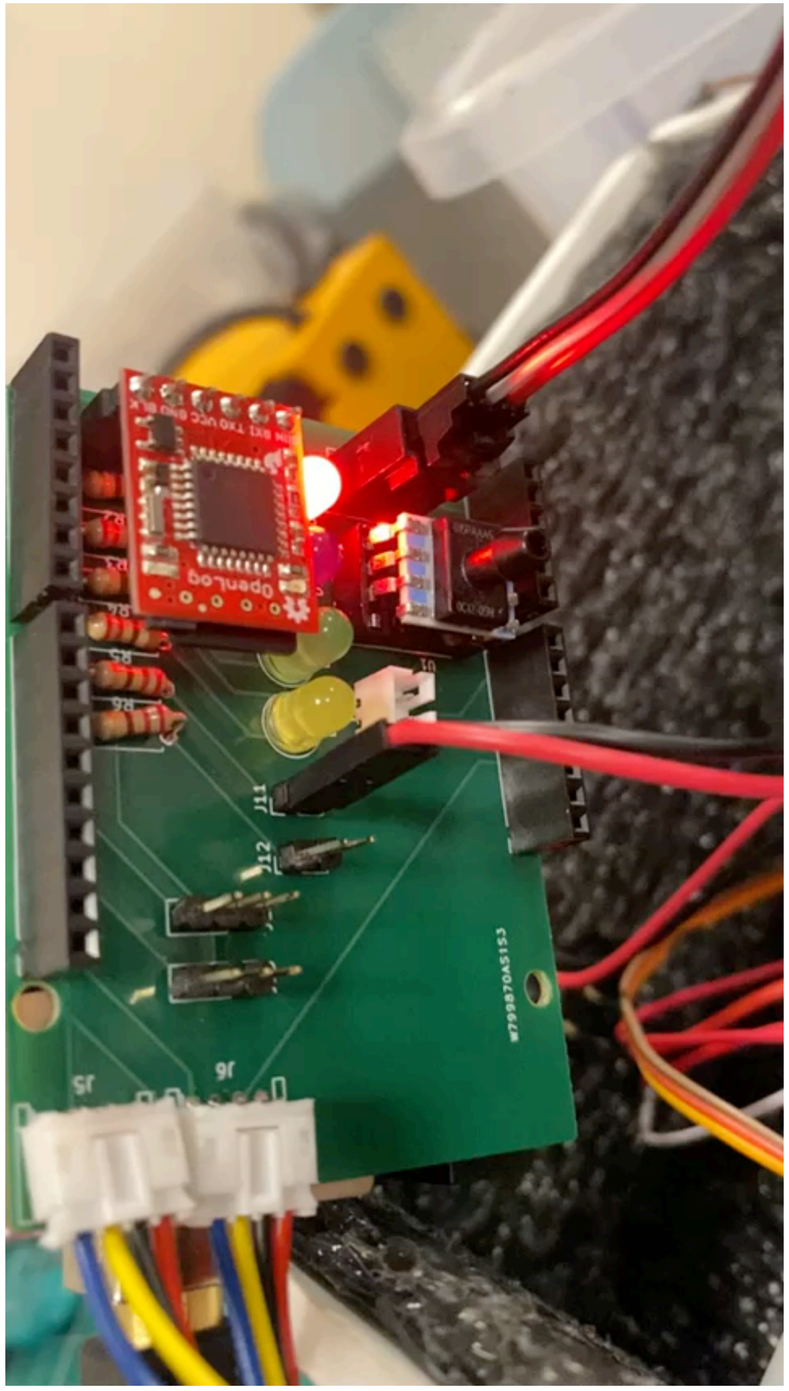


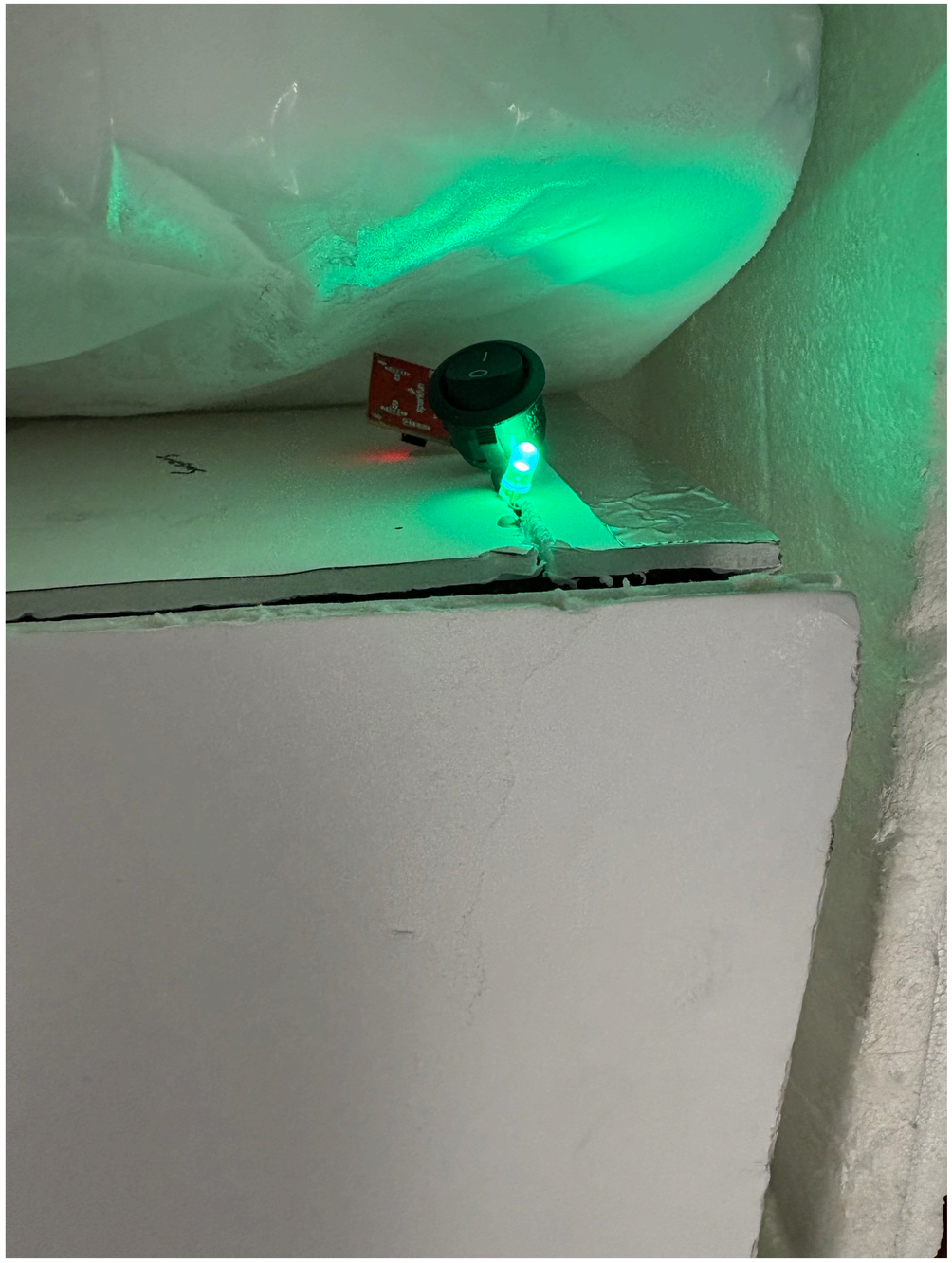




Aaron 5/4/24, 7:03 AM

It got the street glow!

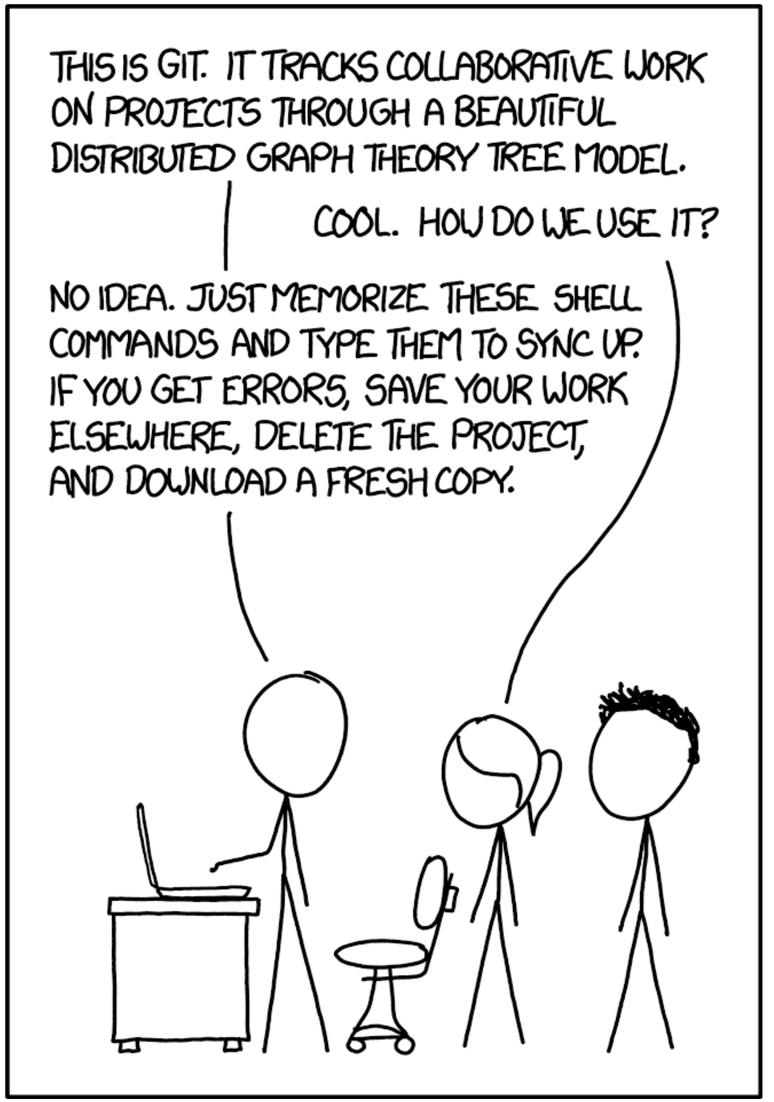




What's on a Student's Computer

and why is it Windows?

- **Every** student is on windows, but probably hasn't set up WSL
- VS Code
  - Extensions?
- Everybody's heard of GitHub, nobody knows git!
  - Discord and Teams are NOT version control!!
- Tooling, compilation, etc can be a real issue



Windows!!

a surprisingly big problem

- **NOT** set up for development, even if the student codes and has taken CS classes
- This means you will encounter problems with:
  - Embedded GCC, MSVC, make, cmake, SSH, USB Devices, USB Drivers, shell scripts, VS Code environments, Environment Variables, Python
- Cargo sidesteps most major pain points
  - `rust-toolchain.toml` and `.cargo/config.toml`

A World of Possibilities

but one semester is not a lot of time!

- Almost every language has some way to run in an embedded context
- C/C++
  - Ubiquitous, Industry Standard, Familiar
  - Some surprisingly large hurdles to clear, even when everyone writes C++
- Circuit/MicroPython
  - Mostly a step in the wrong direction
  - Native modules? Jan Matějek PyCon

Errors

…you will have them

- Indicating errors, especially during setup is arguably the most important thing your payload does
- Let it crash?
- Can you recover gracefully?
- Sometimes your sensor doesn't have data to give you!
- Arduino libraries all handle this differently

```
/*
PROBLEM CODES (AND LED FLASH PATTERN):
  1: I2C ERROR ON 0x73 (flashes yellow)
  2: I2C ERROR ON 0x72 (flashes green)
  3: SERVO CONNECTION FAILURE (all lights flashing at the same time)
  4: EXTERNAL OZONE RUNTIME SENSOR FAILURE (red)
  5: INTERNAL OZONE RUNTIME SENSOR (blue)
  6: OTHER (Red/Blue and Green/Yellow alternating)

pins:
  // RED: 2
  // BLUE: 3
  // GREEN: 4
  // YELLOW: 5
*/
```

```
inline void flashLED(uint32_t pinsToFlash) { // The function used to flash an LED pattern that correlates to one of the problem codes listed at the top
// More memory efficient way of storing LED flash patterns, better than calling a function 20 times a second on this poor little arduino uno.
// All this does is read off bytes of an integer, 1 at a time, and if a byte is 0xFF (a comlpete byte), then flash that LED
  uint8_t firstPin = pinsToFlash & 0xFF;
  uint8_t secondPin = (pinsToFlash >> 8) & 0xFF;
  uint8_t thirdPin = (pinsToFlash >> 16) & 0xFF;
  uint8_t fourthPin = (pinsToFlash >> 24) & 0xFF;

  // RED: 2
  // BLUE: 3
  // GREEN: 4
  // YELLOW: 5
  Serial.print("FLASHING LED: ");
  Serial.println(pinsToFlash, HEX);

  if (firstPin == 0xFF)
    digitalWrite(2, LOW);
  if (secondPin == 0xFF)
    digitalWrite(3, LOW);
  if (thirdPin == 0xFF)
    digitalWrite(4, LOW);
  if (fourthPin == 0xFF)
    digitalWrite(5, LOW);

  delay(FLASH_DELAY);

   if (firstPin > 0)
    digitalWrite(2, HIGH);
  if (secondPin > 0)
    digitalWrite(3, HIGH);
  if (thirdPin > 0)
    digitalWrite(4, HIGH);
  if (fourthPin > 0)
    digitalWrite(5, HIGH);

  delay(FLASH_DELAY);
}
```

```
void loop() {
  if (problemCode > 0) { // Detects if there's a problem code thrown, and flash the accompanying LED pattern
    uint32_t flashCode = 0x00000000;
    switch (problemCode) {
      case 1:
        flashCode |= 0xFF000000;
      break;

      case 2:
        flashCode |= 0x00FF0000;
      break;

      case 3:
        flashCode |= 0xFFFFFFFF;
      break;

      case 4:
        flashCode |= 0x000000FF;
      break;

      case 5:
        flashCode |= 0x0000FFFF;
      break;

      default:
        flashCode = 0xFFFF0000;
        flashLED(flashCode);
        delay(500);
        Serial.println("UNKOWN ERRROR FLASH CODE");
        flashCode = 0x0000FFFF;
      break;
    }
    flashLED(flashCode);
    delay(500);
    if (flashCode != 0xFFFF) { // make an exception for flash code 0xFFFF (Runtime ozone sensors), this is because there's a chance this problem corrects itself and we can continue on with the experiment.
      return;
    }
  }
```

```
// Remember that problem code exception from earlier (0xFFFF)? This is where we detect if a sensor has a problem, or remove the problem code if the problem fixes itself

if (extO3 == 0) {
  EXTsensorRuntimeTries++;
  if (EXTsensorRuntimeTries > 20)
    problemCode = 4;
} else if (extO3 >= 20) {
  EXTsensorRuntimeTries = 0;
  if (problemCode == 4) {
    problemCode = 0;
  }
}

if (intO3 == 0) {
    INTsensorRuntimeTries++;
    if (INTsensorRuntimeTries > 20)
    problemCode = 5;
} else if (intO3 >= 20) {
  INTsensorRuntimeTries = 0;
  if (problemCode == 5) {
    problemCode = 0;
  }
}
```

File structure

We need to add a few things on top of the standard Rust crate since we're cross compiling for a microcontroller.

```
crate
├── target
├── Cargo.lock
├── Cargo.toml
└── src
    └── main.rs
+ + + + +
├── .cargo
│   └── config.toml
├── rust-toolchain.toml
├── build.rs
└── memory.x
```

Two of these files, `rust-toolchain.toml` and `.cargo/config.toml` are for telling our toolchain about our build environment. In each of these we specify a "target triple" which is the string that tells our toolchain which target we're compiling our code for.

- In `.cargo/config.toml` we specify settings for the `cargo` build tool. Since we're cross compiling we need to give it a build target, and sometimes compiler arguments or env variables. We'll also (optionally) specify a 'runner' to automatically compile and flash our binary with `cargo run`

```
[build]
target = "riscv32imc-unknown-none-elf"

runner = "wchisp flash"
```

- Specifying a toolchain in `rust-toolchain.toml` is optional but highly recommended. If you use `rustup` this will tell it which toolchain to automatically download and select for you

```
[toolchain]
channel = "nightly"
targets = ["riscv32imc-unknown-none-elf"]
```

Cargo.toml

Usually the Cargo.toml for a microcontroller firmware doesn't look that different from a typical Rust crate aside from a profile configured with optimizations for size and some unfamiliar dependencies. These crates are designed for a `#![no_std]` context and sometimes make make heavy use of feature flags. Commonly dependencies will include:

- a low level processor access crate
- a hardware abstraction layer (HAL), often implementing traits from embedded-hal or embassy
- sometimes additional embassy or logging/debug crates

```toml
[dependencies]
ch32-hal = { git = "https://github.com/ch32-rs/ch32-hal.git", features = [
    "ch32v203g6u6",
    "memory-x",
    "embassy",
    "rt",
    "time-driver-tim2",
], default-features = false }

# Use same version as ch32-hal
qingke = "*"
qingke-rt = "*"

[profile.release]
strip = false   # Symbols are not flashed to the microcontroller, so don't strip them.
codegen-units = 1
lto = true
opt-level = "z" # Optimize for size.
```

```rust
#![no_std]
#![no_main]

use ch32_hal as hal;
use hal::delay::Delay;
use hal::gpio::{Level, Output};

#[qingke_rt::entry]
fn main() -> ! {
    let p = hal::init(Default::default());
    let mut delay = Delay;

    let mut led = Output::new(p.PA6, Level::Low, Default::default());
    loop {
        led.toggle();
        delay.delay_ms(500);
    }
}

#[panic_handler]
fn panic(_info: &core::panic::PanicInfo) -> ! {
    loop {}
}
```

Clean Code by Default

falling into the *pit of success*

- Nic Barker's Tips for C Programming
  - C99 and `stdint.h`
  - Preprocessor and includes
  - Bounds checking and memory safety
  - Strings, slices, references
  - Lifetimes, allocation, and arenas
- NASA and "The Power of Ten"

Rules [edit]

The ten rules are:[1]

1. Avoid complex flow constructs, such as goto and recursion.
2. All loops must have fixed bounds. This prevents runaway code.
3. Avoid heap memory allocation after initialization.
4. Restrict functions to a single printed page.
5. Use a minimum average of two runtime assertions per function.
6. Restrict the scope of data to the smallest possible.
7. Check the return value of all non-void functions, or cast to void to indicate the return value is useless.
8. Use the preprocessor only for header files and simple macros.
9. Limit pointer use to a single dereference, and do not use function pointers.
10. Compile with all possible warnings active; all warnings should then be addressed before release of the software.

Pico Balloons

Rust at 40,000ft > Rust at 100,000ft

- Ultra low power
  - Solar panels and supercapacitors
- You’re not getting it back so it has to be cheap
- Telemetry is a must
- You might want async (you probably don’t)
- More than enough physical failure modes, software should be reliable

Bonus: Python

Oxidized Python addresses major pain points

- Traditionally we use a matplotlib script for graphing data at the end
- Some random assortment of data libraries, different every time
- Rust plotting libraries are very cool
  - Live data, WASM, more complexity than 2d plots
  - Symposium and Launch on the same day
- UV: Solve the “works on my machine” problem
  - Sometimes it doesn’t even work on their machine!