Advanced Debugging I

Hardware Bring-up, Section Plan

Equipment Required

- 192 car
- Logic analyzer with mini probes, cable
- PC scope with probes, M-F breadboard wire, USB cable
- Voltmeter
- Laptop with mouse, charger
- Plush ducks

Preliminary Discussion

- Scenario: You have a great idea, you designed the hardware, and just finished assembling your board. Now the daunting challenge of writing firmware remains. What do you do?
  - Obviously, write code...
- **How to write code to minimize debugging time?**
  - Write and test small pieces, incrementally
  - If something breaks, in most cases can narrow down culprits easily
  - Make sure basics are sane and solid before building on top
- Obviously, easier said than done
  - So it’s interactive demo day again! Showing you an example of a system bring-up, what can fail, and how to diagnose and deal with failures
- Demo platform introduction
  - Microcontroller-controlled line following robot car
  - Hardware to bring up: LEDs, user rotary encoder, camera, servo, sensors, motor driver
  - Major software blocks: hardware drivers, line detection algorithm, and control loop
  - Today, will be doing basic hardware bring-up and low-level debugging
  - Thursday, will talk about optimization and software structuring

Basic System Bring-up

- Start with mbed peripherals (DigitalOut, DigitalIn, …) defined - make pin mappings
- First thing should be to bring up debugging infrastructure
  - **What are some good options for this?**
  - Should provide feedback to the user about what the MCU is “thinking”
  - printf (UART console), and LEDs “hello, world!"

in main():

    serial.baud(115200);
    serial.printf("Built " __DATE__ " " __TIME__ "\r\n");

    while (1) {
        Led_SR = 1;
        Led_SG = 1;
        Led_SB = 0;
        wait(0.5);
        Led_SR = 0;
        Led_SG = 0;
        Led_SB = 1;
        wait(0.5);
    }

- Now we know the basic system is functional and have a console that can display arbitrary messages
- Bring up the user quad encoder / switch device
  - TODO: add pictures of waveforms
  - Quad encoder: uses two digital channels to detect forward and backward rotation, based on the sequence in which channels polarity. Pretty common device.
  - **How can we tell if the switch works?**
    - (probably either route to LED or printf)
  - **How should we bring up the quad encoder?**
    - It’s a very common device, someone else has probably already written a library for it and many others have probably tested it, so there’s a good chance it works out of the box.
    - Integrate the library and test its behavior. Don’t reinvent the wheel if you don’t need to.
    - I’ve already downloaded it from mbed into my project tree, as well as added it as a library in the build scripts.
```c
#include "qei.h"

QEI UEnc(PTA17, PTA16, NC, 30);

in main():
    while (1) {
        Led_SG = !UEnc_Sw;
        serial.printf("QEI: %i\r\n", UEnc.getPulses());
    }
```

- **Bring up Servo**
  - **Ensure 6v line disconnected (DC/DC converter off, battery disconnected)**
  - **How Servo protocol works**
    - TODO: slides
    - Servos count pulse timing to set position setpoint
    - Internal circuitry adjusts motor current to rotate to setpoint
    - Pulse in range of about 1.0ms (full negative) to 2.0ms (full positive), with center (1.5ms) being center
  - And as might be expected... something this common already has a library, mbed Servo
    - Instantiate over a PWM0-capable pin, write(percentage)
    - **What would be a good sanity check code?**
    - Example: center servo and see if works

```c
#include "Servo.h"

Servo Servo1(PTA5);

in main():
    Servo1.write(0.5);
```

- As you might expect from a debugging discussion, it doesn’t...
- **What are some things that might go wrong? How can we test it?**
- Is the software bad? Is a signal being generated? How to tell? Oscilloscope
  - Measures voltage over time, basically see the waveform on the whiteboard
  - Important parameters: voltage scale (V/div), timescale (s/div), trigger (where it begins acquiring - useful to align repeating signals on - rising or falling edge on a set threshold) or continuous capture, 1x vs. 10x voltage scale (matching probe)
  - Some scopes have autoscale, but you should know what signal to expect, otherwise you could be amplifying noise / a signal that isn’t there
- Less likely: forgot to supply voltage? Also measure that with a scope
- Could have also measured with a voltmeter, but we had the scope handy
- Yeah, oops, do it again with a battery connected

- Bring up the line camera
  - Todo: add picture of datasheet waveforms
  - A line camera: imagine an image from a normal camera, and take a slice through the image - get a 1d array of pixels
  - Communication: no standardized protocol, has its own parallel analog interface
    - Generally true for many parts - simple-ish, nonstandard protocols
    - Clk line shifts out the next pixel value on the analog line
    - Si resets the camera pixel buffers
  - Whiteboard: expected waveform for integrate then sample (can't control loop timing)?
  - Example: write code and test it, but analog sample on both edges to introduce an error

```c
while (1) {
    Cam_clk = 0;
    Cam_si = 1;
    Cam_clk = 1;
    Cam_si = 0;

    for (uint8_t i=0; i<CAMERA_PIXEL_COUNT; i++) {
        Cam_clk = 0;
        Cam_clk = 1;
    }
    wait_us(CAMERA_INTEGRATION_US);

    Cam_clk = 0;
    Cam_si = 1;
    Cam_clk = 1;
    Cam_si = 0;

    float data[CAMERA_PIXEL_COUNT];
    for (uint8_t i=0; i<CAMERA_PIXEL_COUNT; i++) {
        data[i] = Cam_adc;
        Cam_clk = !Cam_clk;
    }
}
```
- Print the array as a string
  
```c
for (uint8_t i=0; i<CAMERA_PIXEL_COUNT; i++) {
    serial.printf("%0.2f", data[i]);
}
serial.printf("\r\n");
```

- Run the code - what should we expect to see here?
  - The output is a hot mess. How do we fix it? How can we compress the data so it’s all on one line?

```c
serial.printf("%1.0f", data[i]*9);
```

- Not what we expect... why? What are all the things that could have gone wrong?
  - Probably want visibility into the system at this point, “see” the waveforms on the camera lines - handy dandy ... oscilloscope!
  - Good thing this board was designed for test, exposed probe hooks on major signal lines
  - Check Aout and clk waveforms
  - See we’re double sampling
  - Fix in code

```c
for (uint8_t i=0; i<CAMERA_PIXEL_COUNT; i++) {
    data[i] = Cam_adc;
    Cam_clk = 1;
    Cam_clk = 0;
}
```

- Bring up I2C analog sensors (mainly thermistors)
  - Ensure SCK wire (white) disconnected
  - Ensure software slightly wrong (protocol slightly screwed up)
  - You’ve (hopefully) already learned about I2C
  - To bring up a new chip, read the protocol on the datasheet
  - TODO: datasheet section on slides
  - With my magic time machine, I magicked up all the ADC chip interface code
#include "ads1015.h"

ADS1015 adc(ext_i2c, ADS1015::ADDR_VDD);

in main():
    serial.printf("ADC: %.02f V\r\n", adc.read_channel_volts(2));

    ... moment of truth ... and it’s wrong (obviously!)
    * Where to begin with debug? What could have gone wrong? What tools to use? *
    * Can scope the destination to see what’s going on there... no signal *
    * That’s not right, use voltmeter in continuity mode *
    * It’s always a connectors issue *
    * Break out a handy dandy ... replacement cable! *
    * Try again, still don’t get the correct data, even though scope *
    * Now we want to analyze the protocol - see if the communications don’t just exist, but are correct *
    * Use a logic analyzer, check against datasheet, and fix *

in ads1015.h
static const uint8_t ADDRESS_BASE = 0x48;

    * And hey, reasonable voltages! *
    * Can also do the volts-to-temperature conversion: *

in main():
    serial.printf("ADC: %.02f C\r\n", (adc.read_channel_volts(2) - 0.4) / 0.0195);

    • For-fun demo
      * Wire it all up to P(ID) and demonstrate it working ... slowly

#include "line_detect.h"
in main():
  
  while (1) {
    Cam_clk = 0;
    Cam_si = 1;
    Cam_clk = 1;
    Cam_si = 0;
    
    for (uint8_t i=0; i<CAMERA_PIXEL_COUNT; i++) {
      Cam_clk = 0;
      Cam_clk = 1;
    }
    wait_us(CAMERA_INTEGRATION_US);
    
    Cam_clk = 0;
    Cam_si = 1;
    Cam_clk = 1;
    Cam_si = 0;
    
    float data[CAMERA_PIXEL_COUNT];
    for (uint8_t i=0; i<CAMERA_PIXEL_COUNT; i++) {
      data[i] = Cam_adc;
      Cam_clk = 0;
      Cam_clk = 1;
    }
    for (uint8_t i=0; i<CAMERA_PIXEL_COUNT; i++) {
      serial.printf("%1.0f", data[i]*10);
    }
    serial.printf("\r\n");
    
    int8_t line_pixel = getLine<CAMERA_PIXEL_COUNT>(1, 16, data);
    if (line_pixel >= 0) {
      float steering_pct = (float)(line_pixel - 64) / 64 * STEERING_KP + 0.5f;
      Servo1.write(steering_pct);
      Led_SR = 0;
      Led_SG = !Led_SG;
    } else {
      Led_SR = !Led_SR;
      Led_SG = 0;
    }
  }
- Obviously won't track a line at any award-winning speed
- But that's next time!

- **Summary**
  - Whirlwind tutorial of bringing up a system
    - A lot of artificial failures that could happen
    - A lot of details glossed over, but the major theme is similar to what you could do
  - Bring things up incrementally
    - Less things added, less things to question if something doesn't work
    - Verify each part
    - Imagine if wrote all the (buggy) code today at once… and nothing worked
      - Where to even start?
  - Get visibility into system if something doesn’t work: measure, measure, measure
    - Right tool for the right job
      - Voltmeter for non-time-varying quantities (voltage rails) and continuity
      - Oscilloscope for analog signals
      - Logic analyzer for protocol analysis or when you want lots of digital inputs
  - Your board will be larger and less purpose built
    - Design for test: headers pins make good test points
    - Don't need mini-grabbers
    - Add more LEDs