Optimization
Recap

We brought up the basic hardware and firmware last time

- Line camera
- Servo
- Line detection and controller loop

But it’s slow

- For a fast car: will lose the line by the time steering updates

How do we optimize?
Optimizing

How much should we optimize?
Or, what other limiting factors are there?
How much should we optimize?
Or, what other limiting factors are there?

- Physical

- Electrical
How much should we optimize?
Or, what other limiting factors are there?

- **Physical**
  - Camera exposure / integration time (50ms)
  - Servo mechanical response (160ms / 60deg)

- **Electrical**
  - Servo update rate (20ms)
Optimizing

How much should we optimize?
Or, what other limiting factors are there?

▶ Physical
  ▶ Camera exposure / integration time (50ms)
  ▶ Servo mechanical response (160ms / 60deg)

▶ Electrical
  ▶ Servo update rate (20ms)

Just for fun, what are some ways around these?
Measuring

How can we measure the update rate? What are some trade-offs?

What takes the most amount of time?
Measuring

How can we measure the update rate? What are some trade-offs?

- `printf` and timers
- Logic analyzer / oscilloscope on external IO, like Serial
- Dedicated timing signals on external IO

What takes the most amount of time?
Overlapped / pipelined operations

How can we optimize this?

camera.read_frame(data); // discard to start integration
wait_ms(50);
camera.read_frame(data); // actually read the frame
float steering = controller.update(line_detect(data)); // expensive
servo.write(steering);
float temperature = read_temperature(); // slow I2C I/O operations
Overlapped / pipelined operations

How can we optimize this?

camera.read_frame(data); // discard to start integration
wait_ms(50);
camera.read_frame(data); // actually read the frame
float steering = controller.update(line_detect(data)); // expensive
servo.write(steering);
float temperature = read_temperature(); // slow I2C I/O operations

Overlap temperature read and line detect with camera integration

timer.reset();
camera.read_frame(data); // actually read the frame
float steering = controller.update(line_detect(data)); // expensive
servo.write(steering);
float temperature = read_temperature(); // slow I2C I/O operations
while (timer.read_ms() < 50); // block until 50ms elapsed

What are the limits of this technique?

▶ Only works when blocking (wait(...)) happens in foreground...
▶ Difficult to write this kind of code...
Overlapped / pipelined operations

How can we optimize this?

```c
camera.read_frame(data);  // discard to start integration
wait_ms(50);
camera.read_frame(data);  // actually read the frame
float steering = controller.update(line_detect(data));  // expensive
servo.write(steering);
float temperature = read_temperature();  // slow I2C I/O operations
```

Overlap temperature read and line detect with camera integration

```c
timer.reset();
camera.read_frame(data);  // actually read the frame
float steering = controller.update(line_detect(data));  // expensive
servo.write(steering);
float temperature = read_temperature();  // slow I2C I/O operations
while (timer.read_ms() < 50) ;  // block until 50ms elapsed
```

What are the limits of this technique?

- Only works when blocking (wait(...)) happens in foreground...
- Difficult to write this kind of code...
Threads abstraction

Manually sequencing operations is *hard*. Why not have the computer handle concurrency with threads?

```c
void camera_thread() {
    float data[128];
    camera.read_frame(data);
    // intensive compute
    float steering = controller.update(line_detect(data));
    servo.write(steering);
    // integrate and yield
    Thread::wait_ms(50);
}
```

```c
void temperature_thread() {
    // slow I2C I/O operations
    float temperature = read_temperature();
    // ... do something here ...
    // rate limiting
    Thread::wait_ms(50);
}
```

*What could possibly go wrong?*
Threads abstraction

Manually sequencing operations is *hard*. Why not have the computer handle concurrency with threads?

```c
void camera_thread() {
    float data[128];
camera.read_frame(data);
    // intensive compute
float steering = controller.
        update(line_detect(data));
servo.write(steering);
    // integrate and yield
Thread::wait_ms(50);
}
```

```c
void temperature_thread() {
    // slow I2C I/O operations
float temperature =
        read_temperature();
    // ... do something here ...
    // rate limiting
Thread::wait_ms(50);
}
```

*What could possibly go wrong?*

- Hidden limits (jiffy interval, thread switch latency, overhead)
- Non-hard-realtime (timing not guaranteed) RTOS
- Race conditions, deadlock, starvation, livelock, ...

An RTOS is a powerful tool, but comes with caveats. Use with care!
Even with RTOS / overlapping, most bulk IO operations are blocking. What do?
DMA architectures

Even with RTOS / overlapping, most bulk IO operations are blocking. What do?

**DMA: Direct Memory Access**
- Automatically transfer blocks of data between memory or peripherals
- Typically: set up transfer (src, dst, length), then start
  - Operates without CPU intervention

... but mbed support for DMA is lacking and nonstandard.
- If you need this, you will have to register poke
- Details likely in chip datasheet / reference manual
What if we didn’t have a linear thermistor or FPU?

```c
float volts_to_temperature(float volts) {
    const float R_REF = 10000; // 10k thermistor
    const float BETA = 3428, R_INF = R_REF * exp(-BETA / 298.15);
    const float V_REF = 3.3; // high voltage reference
    float resistance = (R_REF * V_REF / volts) - R_REF;
    float temp = BETA / log((resistance) / R_INF);
    return temp - 273.15; // degK to degC
}
```

This is really, really expensive.

How can this be optimized?
Compute optimizations

What if we didn’t have a linear thermistor or FPU?

```c
float volts_to_temperature(float volts) {
    const float R_REF = 10000; // 10k thermistor
    const float BETA = 3428, R_INF = R_REF * exp(-BETA / 298.15);
    const float V_REF = 3.3; // high voltage reference
    float resistance = (R_REF * V_REF / volts) - R_REF;
    float temp = BETA / log((resistance) / R_INF);
    return temp - 273.15; // degK to degC
}
```

This is really, really expensive.

How can this be optimized?

- Replace with fixed point operations
- Lookup tables: precompute values before runtime
  - Will need to discretize input, for example:
    ```c
    volts_to_temperature(adc.read_u16())
    ```
Cost of abstraction: DigitalOut

When you do this,

```plaintext
DigitalOut my_led (PTC8);
...
my_led = 1;
```

What actually goes on behind the scenes?
Cost of abstraction: Under the hood

```c
#include <mbed_api/DigitalOut.h>

class DigitalOut {
    void write(int value) {
        gpio_write(&gpio, value);
    }
}
```

```c
#include <mbed_targets/hal/TARGET_Freescale/TARGET_KLXX/gpio_object.h>

typedef struct {
    PinName pin;
    uint32_t mask;
    __IO uint32_t *reg_dir;
    __IO uint32_t *reg_set;
    __IO uint32_t *reg_clr;
    __I uint32_t *reg_in;
} gpio_t;

static inline void gpio_write(gpio_t *obj, int value) {
    if (value)
        *obj->reg_set = obj->mask;
    else
        *obj->reg_clr = obj->mask;
}
```

Lots of indirection to write a register! What’s the trade-off?
Cost of abstraction: Under the hood

Lots of indirection to write a register! What’s the trade-off?

- Much slower performance, but readable and portable code
Cost of abstraction: Tip of the iceberg

When you do this,

```c
AnalogIn my_adc(PTC2);
...
float adc_val = my_adc;
```

What decisions are implicitly made for you?
Cost of abstraction: Tip of the iceberg

When you do this,

```c
AnalogIn my_adc(PTC2);
... 
float adc_val = my_adc;
```

What decisions are implicitly made for you?

- All the ADC configurations
- Like bit accuracy, speed, power settings, and much more ...

Probably sufficient for most users, but you aren’t getting all your hardware can do

What do you gain?

- Portability across platforms
- Simplicity of setup

MKL25Z ADC registers
Optimization
Abstractions

Cost of abstraction: Tip of the iceberg

When you do this,

```c
AnalogIn my_adc(PTC2);
...
float adc_val = my_adc;
```

What decisions are implicitly made for you?

- All the ADC configurations
- Like bit accuracy, speed, power settings, and much more ...

Probably sufficient for most users, but you aren’t getting all your hardware can do

What do you gain?

- Portability across platforms
- Simplicity of setup

MKL25Z ADC registers
Software Engineering
Why?

What are our goals in writing firmware?

- Correct
- Maintainable - when you need to fix something
- Extensible - because feature creep happens

All related: unmaintainable, unreadable code tends to hide bugs!

Let's see why!
Why?

What are our goals in writing firmware?

- Correct
- Maintainable - when you need to fix something
- Extensible - because feature creep happens

All related: unmaintainable, unreadable code tends to hide bugs!

Let’s see why!
Software Engineering  Antipatterns

Two Cameras

Say I modularized the camera reading into a class:

```c
Camera camera1 (PTB2 /* CLK */, PTB3 /* SI */, PTC2 /* AO */);
```

```c
void control_loop () {
    float * data = camera1.read_frame ();
    float steering = controller.update(line_detect(data));
    servo.write(steering);
    wait_ms(50);
}
```

Given the structure, how would I add another camera?
Two Cameras

Say I modularized the camera reading into a class:

```c
Camera camera1(PTB2 /*CLK*/, PTB3 /*SI*/, PTC2 /*AO*/);
```

```c
void control_loop() {
    float *data = camera1.read_frame();
    float steering = controller.update(line_detect(data));
    servo.write(steering);
    wait_ms(50);
}
```

Given the structure, how would I add another camera?

- Simple, right? Instantiate another Camera?
  - Camera camera2(PTB4, PTB5, PTC1);
Say I modularized the camera reading into a class:

```c
Camera camera1(PTB2 /*CLK*/, PTB3 /*SI*/, PTC2 /*AO*/);
```

```c
void control_loop() {
    float *data = camera1.read_frame();
    float steering = controller.update(line_detect(data));
    servo.write(steering);
    wait_ms(50);
}
```

Given the structure, how would I add another camera?

- Simple, right? Instantiate another Camera?
  - Camera camera2(PTB4, PTB5, PTC1);

What hidden assumptions / expectations did I have for Camera?
What if the Camera implementation looked like this?

```cpp
uint16_t camera_data[CAMERA_PIXELS]; // global

class Camera {
public:
  Camera(PinName clk, PinName si, PinName adc);
  float* read_frame() {
    /* ADC reads into global camera_data */
  }
}
```
What if the Camera implementation looked like this?

```cpp
uint16_t camera_data[CAMERA_PIXELS]; // global
class Camera {
    public:
        Camera(PinName clk, PinName si, PinName adc);
        float* read_frame() {
            /*ADC reads into global camera_data*/
        }
}
```

Ruh roh

- Globals break encapsulation provided by objects
- ... and this breaks user expectations of independence
- DON’T DO IT!
While we’re talking about globals, what anti-patterns can arise from this?

```c
float motor_velocity_target; // global

void main() {
    motor_velocity_target = 3.0;
    // rest of code here
}
```

So far, so good, right?
While we’re talking about globals, what anti-patterns can arise from this?

```c
float motor_velocity_target; // global

void main () {
    motor_velocity_target = 3.0;
    // rest of code here
}
```

So far, so good, right?

Perhaps I also have a kill switch in another function:

```c
if (kill_switch) motor_velocity_target = 0;
```

Soon, you have no clue what the target actually is - dataflow spaghetti!
While we’re talking about globals, what anti-patterns can arise from this?

```c
float motor_velocity_target; // global

void main () {
    motor_velocity_target = 3.0;
    // rest of code here
}
```

So far, so good, right?

Perhaps I also have a kill switch in another function:
```
if (kill_switch) motor_velocity_target = 0;
```
And why not have it dependent on tracking, perhaps in a different .c file:
```
if (bad_tracking) motor_velocity_target -= 0.1;
```
Globals

While we’re talking about globals, what anti-patterns can arise from this?

```c
float motor_velocity_target; // global

void main() {
    motor_velocity_target = 3.0;
    // rest of code here
}
```

So far, so good, right?

Perhaps I also have a kill switch in another function:

```c
if (kill_switch) motor_velocity_target = 0;
```

And why not have it dependent on tracking, perhaps in a different .c file:

```c
if (bad_tracking) motor_velocity_target -= 0.1;
```

Soon, you have no clue what the target actually is - dataflow spaghetti!
Let’s take another look at the camera protocol and line detection

Can you **easily** tell what this code does?

```c
// in main() loop
si = 1; si = 0;
float data[128];
for (int i = 0; i < 128; i++) {
    clk = 0; clk = 1;
    data[i] = ain;
}
float max = 0; uint8_t pos = 0;
for (int i = 0; i < 128; i++) {
    if (data[i] > max) {
        max = data[i]; pos = i;
    }
}
servo.write((float)pos / 128);
```
Let’s take another look at the camera protocol and line detection

Can you **easily** tell what this code does?

```c
// in main() loop
si = 1; si = 0;
float data[128];
for (int i=0; i<128; i++) {
    clk = 0; clk = 1;
    data[i] = ain;
}
float max = 0; uint8_t pos = 0;
for (int i=0; i<128; i++) {
    if (data[i] > max) {
        max = data[i]; pos = i;
    }
}
servo.write((float)pos / 128);
```

Probably not.
Oh Dear...

Is this better? Why?

```c
const uint8_t CAMERA_LENGTH = 128, CAMERA_HALF = CAMERA_LENGTH / 2;
void camera_read(float* data_out) {
    si = 0; si = 0;
    for (int i=0; i<CAMERA_LENGTH; i++) {
        clk = 0; clk = 1;
        data_out[i] = ain;
    }
}

uint8_t line_detect(float* cam_data) {
    float max = 0; uint8_t pos = 0;
    for (int i=0; i<CAMERA_LENGTH; i++) {
        if (cam_data[i] > max) {
            max = cam_data[i]; pos = i;
        }
    }
    return pos;
}

// in main() loop
float cam_data[CAMERA_LENGTH];
camera_read(cam_data);
int8_t line_offset = CAMERA_HALF - line_detect(cam_data);
servo.write((float)line_offset/CAMERA_LENGTH);
```
Good Programming Style

Good style produces readable and maintainable code, saving you time later

- Short functions, single responsibility
  - Make it easy to understand
- Consistent level of abstraction
  - Separate the “what” from the “how”
- Don’t repeat yourself (DRY)
  - Copypaste code is bad: making consistent changes becomes very hard
The Old Fashioned Way

Here’s a really basic lost line algorithm:

```c
uint16_t last_line_pos = 0;
motor = 0.7;
while (1) {
    int16_t line_pos = line_detect(camera_data);
    if (line_pos != -1) { // line detected - follow it
        servo.write((float) line_pos / CAMERA_LENGTH);
    } else { // line not found - rail servo in previous direction
        if (last_line_pos < 64) {
            servo.write(0.0);
        } else {
            servo.write(1.0);
        }
        motor = 0.4; // slow down
    }
    last_line_pos = line_pos;
}
```

Is it correct?
The Old Fashioned Way

Here’s a really basic lost line algorithm:

```c
uint16_t last_line_pos = 0;
motor = 0.7;
while(1) {
    int16_t line_pos = line_detect(camera_data);
    if (line_pos != -1) { // line detected - follow it
        servo.write((float)line_pos/CAMERA_LENGTH);
    } else { // line not found - rail servo in previous direction
        if (last_line_pos < 64) {
            servo.write(0.0);
        } else {
            servo.write(1.0);
        }
    }
    motor = 0.4; // slow down
}
last_line_pos = line_pos;
```

Is it correct?  Nope

- `last_line_pos` immediately clobbered, but not obvious at-a-glance
- Implicit state in motor PWM - forget to reset motor to full speed
Let’s make things clearer by following the state machine model:

**Write the transition function**

```c
enum State { FOUND, LOST_LEFT, LOST_RIGHT };

State do_transition(State current_state, int16_t line_pos , int16_t last) {
    if (current_state == FOUND) {
        if (line_pos == -1) {
            if (last <= 64) {
                return LOST_LEFT;
            } else {
                return LOST_RIGHT;
            }
        } else {
            return FOUND;
        }
    } else {
        if (line_pos != -1) {
            return FOUND;
        }
    }
}
```

lost track state machine

graphical notation
With State Machines

Let’s make things clearer by following the state machine model:

**Write the state actions**

```c
enum State { FOUND, LOST_LEFT, LOST_RIGHT };

void state_action(State state, int16_t line_pos, int16_t & last_out) {
    if (state == FOUND) {
        servo.write((float)line_pos/CAMERA_LENGTH);
        motor = 0.7;
        last = line_pos;
    } else if (state == LOST_LEFT) {
        servo.write(0.0);
        motor = 0.4;
    } else if (state == LOST_RIGHT) {
        servo.write(1.0);
        motor = 0.4;
    }
}
```

lost track state machine graphical notation
Let’s make things clearer by following the state machine model:

... and put it all together

```c
int16_t last = 0;
State state = FOUND;
while (1) {
    int16_t line_pos = line_detect(camera_data);
    state = do_transition(state, line_pos, last);
    state_action(state, line_pos, last);
}
```

lost track state machine
graphical notation
What could *possibly* go wrong?

If your firmware was to fail in operation, what would the consequences be?
What could *possibly* go wrong?

If your firmware was to fail in operation, what would the consequences be?

What if you were building man-rated spacecraft, or a nuclear reactor?
What could *possibly* go wrong?

If your firmware was to fail in operation, what would the consequences be?

What if you were building man-rated spacecraft, or a nuclear reactor?

How might you approach writing firmware differ for safety-critical systems?
What could *possibly* go wrong?

If your firmware was to fail in operation, what would the consequences be?

What if you were building man-rated spacecraft, or a nuclear reactor?

How might you approach writing firmware differ for safety-critical systems?

▶ Lower complexity
▶ Better documented code
▶ Code reviews
▶ More testing and/or static analyzers
▶ .. and much more ...

Many style guidelines exist for safety-critical embedded code. Power of 10 (https://spinroot.com/p10) is one.
Flow control constructs  Power of Ten #1

In a less civilized day and age:

```c
void update_outputs() {
    float motor_speed = 1;
    if (lost_line) goto low_speed;
    if (bad_line_quality) goto fault;
    goto okay;

done:
    motor = motor_speed;
    return;

low_speed:
    motor_speed = 0.5;

fault:
    red_led = 1;
    goto done;

okay:
    red_led = 0;
    goto done;
}
```

What does this code do?
Is it easy to understand?
Flow control constructs  Power of Ten #1

In a less civilized day and age:

```c
void update_outputs () {
    float motor_speed = 1;
    if ( lost_line ) goto low_speed;
    if ( bad_line_quality ) goto fault;
    goto okay;

done:
    motor = motor_speed;
    return;

low_speed:
    motor_speed = 0.5;
fault:
    red_led = 1;
    goto done;
okay:
    red_led = 0;
    goto done;
}
```

What does this code do?

Is it easy to understand?

Is this better? Why?

```c
void update_outputs() {
    if ( lost_line ) {
        motor = 0.5;
        red_led = 1;
    } else if ( bad_line_quality ) {
        motor = 1;
        red_led = 1;
    } else {
        motor = 1;
        red_led = 0;
    }
}
```
Flow control constructs

In a less civilized day and age:

```c
void update_outputs () {
    float motor_speed = 1;
    if (lost_line) goto low_speed;
    if (bad_line_quality) goto fault;
    goto okay;

done:
    motor = motor_speed;
    return;

low_speed:
    motor_speed = 0.5;

fault:
    red_led = 1;
    goto done;

okay:
    red_led = 0;
    goto done;
}
```

Is this better? Why?

```c
void update_outputs() {
    if (lost_line) {
        motor = 0.5;
        red_led = 1;
    } else if (bad_line_quality) {
        motor = 1;
        red_led = 1;
    } else {
        motor = 1;
        red_led = 0;
    }
}
```

goto makes control paths confusing
Avoid using these constructs

What does this code do?
Is it easy to understand?
What could go wrong with this code?

```c
float read_accelerometer() {
    while (accelerometer.data_available() != accelerometer::READY);
    return accelerometer.read();
}

void main() {
    while (1) {
        float acceleration = read_accelerometer();
        red_led = abs(acceleration) > 2;
        do_some_safety_critical_update();
    }
}
```
What could go wrong with this code?

```c
float read_accelerometer() {
    while (accelerometer.data_available() != accelerometer::READY);  
    return accelerometer.read();
}

void main() {
    while (1) {
        float acceleration = read_accelerometer();
        red_led = abs(acceleration) > 2;
        do_some_safety_critical_update();
    }
}

You can probably guess the accelerometer isn’t critical
But do_somesome_safety_critical_update() looks critical.
What if the accelerometer failed during operation?
What could go wrong with this code?

```c
float read_accelerometer() {
    while (accelerometer.data_available() != accelerometer::READY);
    return accelerometer.read();
}

void main() {
    while (1) {
        float acceleration = read_accelerometer();
        red_led = abs(acceleration) > 2;
        do_some_safety_critical_update();
    }
}
```

You can probably guess the accelerometer isn’t critical
But `do_some_safety_critical_update()` looks critical.
What if the accelerometer failed during operation?

- System stall! Safety updates don’t happen!
- Yaaaaayyyy! *(not!)*

Be wary of unbounded loop iterations, prevent runaway code.
What could go wrong with this code?

```c
float* read_camera() {
    float* data = (float*)malloc(128*sizeof(float));
    for (uint8_t i=0; i<128; i++) {
        // integration and pixel shifting control here
        data[i] = cam_adc;
    }
    return data;
}

void main() {
    while (1) {
        float* data = read_camera();
        servo.write((float)line_detect(data)/128);
    }
}
```

Looks pretty, right? But what lurks underneath?
What could go wrong with this code?

```c
float* read_camera() {
    float* data = (float*)malloc(128*sizeof(float));
    for (uint8_t i = 0; i < 128; i++) {
        // integration and pixel shifting control here
        data[i] = cam_adc;
    }
    return data;
}

void main() {
    while (1) {
        float* data = read_camera();
        servo.write((float)line_detect(data)/128);
    }
}
```

Looks pretty, right? But what lurks underneath?
What could go wrong with this code?

```c
float* read_camera() {
    float* data = (float*) malloc(128* sizeof(float));
    for (uint8_t i = 0; i < 128; i++) {
        // integration and pixel shifting control here
        data[i] = cam_adc;
    }
    return data;
}

void main() {
    while (1) {
        float* data = read_camera();
        servo.write((float) line_detect(data)/128);
    }
}
```

Looks pretty, right? But what lurks underneath?

- `malloc() without free()` - memory leak
- `malloc()` result not checked for failure
- Difficult to prove program fits in memory constraints
- Potential issues with memory fragmentation

Recommended to avoid dynamic allocation
You already know what I’m going to ask.

```c
float* average_samples(float* data, size_t samples) {
    float sum = 0;
    for (size_t i=0; i<samples; i++) {
        sum += data[i];
    }
    return sum / samples;
}
```
You already know what I’m going to ask.

```c
float* average_samples(float* data, size_t samples) {
    float sum = 0;
    for (size_t i = 0; i < samples; i++) {
        sum += data[i];
    }
    return sum / samples;
}

NaN NaN NaN NaN NaN NaN NaN NaN
```
You already know what I’m going to ask.

```c
float* average_samples(float* data, size_t samples) {
    float sum = 0;
    for (size_t i=0; i<samples; i++) {
        sum += data[i];
    }
    return sum / samples;
}
```

NaN NaN NaN NaN NaN NaN NaN … Batman!

- Potential division by zero, but returns a NaN instead of failing
- No helpful debugging message, just strange behavior

`assert()` can be used to test pre-conditions (like `samples > 0`) and fail noisily or initiate recovery actions.
4, 6, and 7 mainly deal with code style

- We’ve already covered some style examples

8 limits preprocessor usage

- Preprocessor is based on string substitution and had complex features

9 limits pointer complexity

- Multiple levels of pointer indirection is *hard*!

10 encourages use of static analysis tools and compiling with all warnings

- ... because a computer is more thorough than a human
Security and Privacy

So Internet of Things is really hot right now ...
So Internet of Things is really hot right now ... some might even say,

*it's on fire*
Security and Privacy

So Internet of Things is really hot right now ... some might even say, *it's on fire*

What additional concerns are there with Internet-connected devices?
Security and Privacy

So Internet of Things is really hot right now ... some might even say,

*it's on fire*

What additional concerns are there with Internet-connected devices?

- Hacking, to become part of a botnet or for data exfiltration
  - Common exploits: outdated firmware, default passwords, backdoors, ...
- Privacy leaks of user / sensitive data
  - Is data encrypted in transit? Is robust authorization in place?
Optimizations exist if your application is timing sensitive
  ▶ Know what to optimize to: no more, no less
  ▶ I/O probably is the most time consuming operation
Write good code now so you don’t kick yourself later
  ▶ Understand what your code is doing
  ▶ Write clear, concise, readable code
  ▶ Guidelines may be helpful for safety-critical applications
Be cognizant of the full impact of your software and device
  ▶ Criticality
  ▶ Security
  ▶ Privacy