EECS 192: Mechatronics Design Lab

Discussion 11: Embedded Software

written by: Richard "Ducky" Lin Spring 2015

8 & 9 April 2015 (Week 11)

- Multitasking Models
- Software Engineering
- Convenience vs. Performance

Multitasking Models

Motivation

Good cars need simultaneous velocity and steering control

- Velocity control needs to time encoder transitions and set motor PWM
- Steering control needs to wait for camera integration, detect line, and update servo
- Also want to stream telemetry data

Cooperative Multitasking: Example

A simple way to achieve multitasking with an event loop:

```
void main() {
  while (1) {
    if (Camera.is_integration_finished()) {
      Servo.set_steering(Camera.detect_line());
      Camera.restart_integration();
    }
    if (Encoder.is_transition()) {
      SpeedSensor.update(Encoder.get_last_width());
      Motor.set_pwm(TARGET_SPEED - SpeedSensor.get());
    }
    Telemetry.do_io();
  }
}
```

What are some issues? Especially related to timing and correctness?

Cooperative Multitasking: Example

A simple way to achieve multitasking with an event loop:

```
void main() {
  while (1) {
    if (Camera.is_integration_finished()) {
      Servo.set_steering(Camera.detect_line());
      Camera.restart_integration();
    }
    if (Encoder.is_transition()) {
      SpeedSensor.update(Encoder.get_last_width());
      Motor.set_pwm(TARGET_SPEED - SpeedSensor.get());
    }
    Telemetry.do_io();
  }
}
```

What are some issues? Especially related to timing and correctness?

- If camera line detection is too long, may miss encoder transitions
  - Even non-critical telemetry can block critical control operations
- Complex, interleaved control structures hinder readability

Interrupts

So I need some way to ensure critical events aren't missed: Interrupts!

- Hardware functionality which interrupts the CPU on some event (like input transition)
- Saves current position in code, then jumps to the ISR (interrupt service routine)
- Once ISR returns, restore previous position in code and continue executing

Interrupts: Example

Let's handle encoders with an interrupt!

```
void encoder_isr() {
  speed = calculate_speed(EncoderTimer.read_us());
  EncoderTimer.reset();
}
void main() {
  EncoderInterrupt.fall(encoder_isr);
  while (1) {
    wait(CAMERA_INTEGRATION_TIME);
    Servo.set_steering(Camera.detect_line());
    Motor.set_pwm(TARGET_SPEED - speed);
    Telemetry.do_io();
  }
}
```

What did we gain?

Interrupts: Example

Let's handle encoders with an interrupt!

```
void encoder_isr() {
  speed = calculate_speed(EncoderTimer.read_us());
  EncoderTimer.reset();
}
void main() {
  EncoderInterrupt.fall(encoder_isr);
  while (1) {
    wait(CAMERA_INTEGRATION_TIME);
    Servo.set_steering(Camera.detect_line());
    Motor.set_pwm(TARGET_SPEED - speed);
    Telemetry.do_io();
  }
}
```

What did we gain?

- Simpler control logic: camera is just integrate-wait-read
- All encoder transitions recorded, even if faster than camera reads

Interrupts: Example

Let's handle encoders with an interrupt!

```
void encoder_isr() {
  speed = calculate_speed(EncoderTimer.read_us());
  EncoderTimer.reset();
}
void main() {
  EncoderInterrupt.fall(encoder_isr);
  while (1) {
    wait(CAMERA_INTEGRATION_TIME);
    Servo.set_steering(Camera.detect_line());
    Motor.set_pwm(TARGET_SPEED - speed);
    Telemetry.do_io();
  }
}
```

What new issues did we cause?

Interrupts: Example

Let's handle encoders with an interrupt!

```
void encoder_isr() {
  speed = calculate_speed(EncoderTimer.read_us());
  EncoderTimer.reset();
}
void main() {
  EncoderInterrupt.fall(encoder_isr);
  while (1) {
    wait(CAMERA_INTEGRATION_TIME);
    Servo.set_steering(Camera.detect_line());
    Motor.set_pwm(TARGET_SPEED - speed);
    Telemetry.do_io();
  }
}
```

What new issues did we cause?

- Motor controller frequency tied to camera
- `encoder_isr` can fire anytime/anywhere, even interfering with `main`
  - Really bad things can happen if `encoder_isr` is slow
- Potential race conditions with shared variables (like `speed`)

Threading

What if I want to decouple the motor control loop from the camera control loop?

Threads: sequences of instructions managed independently by a scheduler

- Conceptually runs in parallel, but actually time-multiplexed onto CPU
- Threads regularly pre-empted: paused so another thread can run
  - Called a context switch

Threading: Example

Rewriting the same code with threads:

```
void encoder_isr(); // same as previously
void camera_loop() {  // in a while(1) {...} in own thread
  wait(CAMERA_INTEGRATION_TIME);
  Servo.set_steering(Camera.detect_line());
}
void motor_loop() {  // in a while(1) {...} in own thread
  Motor.set_pwm(TARGET_SPEED - SpeedSensor.get());
  wait(MOTOR_UPDATE_TIME);
}
void telemetry_loop() {  // in a while(1) {...} in own thread
  Telemetry.do_io();
}
```

What got better?

Threading: Example

Rewriting the same code with threads:

```
void encoder_isr(); // same as previously
void camera_loop() {  // in a while(1) {...} in own thread
  wait(CAMERA_INTEGRATION_TIME);
  Servo.set_steering(Camera.detect_line());
}
void motor_loop() {  // in a while(1) {...} in own thread
  Motor.set_pwm(TARGET_SPEED - SpeedSensor.get());
  wait(MOTOR_UPDATE_TIME);
}
void telemetry_loop() {  // in a while(1) {...} in own thread
  Telemetry.do_io();
}
```

What got better?

- Code is much cleaner: steering and motor control independent
- Motor update rate independent of camera integration time

Threading: Example

Rewriting the same code with threads:

```
void encoder_isr(); // same as previously
void camera_loop() {  // in a while(1) {...} in own thread
  wait(CAMERA_INTEGRATION_TIME);
  Servo.set_steering(Camera.detect_line());
}
void motor_loop() {  // in a while(1) {...} in own thread
  Motor.set_pwm(TARGET_SPEED - SpeedSensor.get());
  wait(MOTOR_UPDATE_TIME);
}
void telemetry_loop() {  // in a while(1) {...} in own thread
  Telemetry.do_io();
}
```

What issues arise?

Threading: Example

Rewriting the same code with threads:

```
void encoder_isr(); // same as previously
void camera_loop() {  // in a while(1) {...} in own thread
  wait(CAMERA_INTEGRATION_TIME);
  Servo.set_steering(Camera.detect_line());
}
void motor_loop() {  // in a while(1) {...} in own thread
  Motor.set_pwm(TARGET_SPEED - SpeedSensor.get());
  wait(MOTOR_UPDATE_TIME);
}
void telemetry_loop() {  // in a while(1) {...} in own thread
  Telemetry.do_io();
}
```

What issues arise?

- Threads can be pre-empted anywhere, even during camera read
- Thread timing granularity can cause integration time inaccuracy
- Scheduling overhead: context switches take time
- Data sharing could be more complicated, requiring synchronization

Benchmarking

But just how bad are those issues?
More importantly, how can we tell?

Benchmarking

But just how bad are those issues?
More importantly, how can we tell?

Benchmark time, of course!

- Want to determine context switch overhead and schedule frequency
- Strategy
  - Instantiate some threads
  - Each rapidly toggles IO, indicating running
  - View each thread's IO on scope

Benchmarking

But just how bad are those issues?
More importantly, how can we tell?

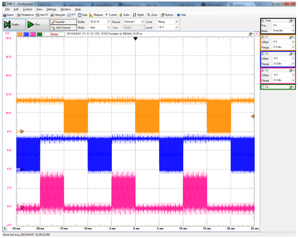
Benchmark time, of course!

- Want to determine context switch overhead and schedule frequency
- Strategy
  - Instantiate some threads
  - Each rapidly toggles IO, indicating running
  - View each thread's IO on scope

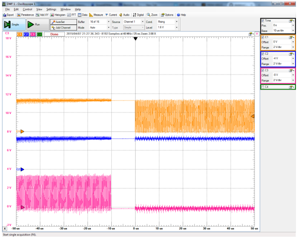
Results:

- Scheduler invocation every 5ms
- Context switch overhead is about 10us

So, this could really mess with integration time.



measure frequency: 5 ms/div



measure overhead: 10 us/div

Better Camera Timing

A simple solution to meet realtime constraints is to change priorities:

```
void camera_thread_fn() {
  while(1) {
    wait(CAMERA_INTEGRATION_TIME);
    Servo.set_steering(Camera.detect_line());
  }
}
void main() {
  ...
  Thread camera_thread(camera_thread_fn);
  camera_thread.set_priority(osPriorityHigh);
  ...
}
```

Why won't this work?

Better Camera Timing

A simple solution to meet realtime constraints is to change priorities:

```
void camera_thread_fn() {
  while(1) {
    wait(CAMERA_INTEGRATION_TIME);
    Servo.set_steering(Camera.detect_line());
  }
}
void main() {
  ...
  Thread camera_thread(camera_thread_fn);
  camera_thread.set_priority(osPriorityHigh);
  ...
}
```

Why won't this work?

- `wait` is a dumb spin loop, won't yield control to lower priority threads
  - Since `camera_thread_fn` never sleeps, other threads "starve"
  - Instead, use `Thread::wait` to yield to other threads

Misc mbed RTOS topics

- Tickers regularly calls functions using ISRs
  - Standard ISR caveats apply
- RtosTimer can also regularly call functions
  - All timers are handled in a single thread, `osTimerThread`
- The default max number of threads is 6
  - `OS_TASKCNT` and other constants in `mbed-rtos/rtx/RTX_Conf_CM.c`

See the mbed RTOS documentation:

`https://developer.mbed.org/handbook/RTOS`

Software Engineering

Oh Dear...

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

```
// in main() loop
si = 1; si = 0;
uint16_t data[128];
for (int i=0; i<128; i++) {
  clk = 0;  clk = 1;
  data[i] = ain.read_u16();
}
uint16_t max = 0; uint8_t pos = 0;
for (int i=0; i<128; i++) {
  if (data[i] > max) {
    max = data[i]; pos = i;
  }
}
servo.write(0.075 + 0.025 * (64.0 - pos) / 64);
```

Oh Dear...

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

```
// in main() loop
si = 1; si = 0;
uint16_t data[128];
for (int i=0; i<128; i++) {
  clk = 0;  clk = 1;
  data[i] = ain.read_u16();
}
uint16_t max = 0; uint8_t pos = 0;
for (int i=0; i<128; i++) {
  if (data[i] > max) {
    max = data[i]; pos = i;
  }
}
servo.write(0.075 + 0.025 * (64.0 - pos) / 64);
```

Probably not.

Oh Dear...

Is this better? Why?

```
const uint8_t CAMERA_LENGTH = 128, CAMERA_HALF = CAMERA_LENGTH / 2;
void camera_read(uint16_t* data_out) {
  si = 0; si = 0;
  for (int i=0; i<CAMERA_LENGTH; i++) {
    clk = 0;  clk = 1;
    data_out[i] = ain.read_u16();
  }
}
uint8_t line_detect(uint16_t* cam_data) {
  uint16_t 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;
}
void set_steering_pct(float pct) {
  servo.write(0.075 + 0.025 * (pct));
}

// in main() loop
uint16_t cam_data[CAMERA_LENGTH];
camera_read(cam_data);
int8_t line_offset = CAMERA_HALF - line_detect(cam_data);
set_steering_pct((float)line_offset/CAMERA_HALF);
```

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

Want to know more? Take cs169!

The Old Fashioned Way

Here's a really basic lost line algorithm:

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

Is it correct?

The Old Fashioned Way

Here's a really basic lost line algorithm:

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

Is it correct? Nope

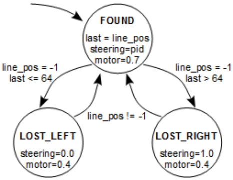
- `last_line_pos` immediately clobbered, but not obvious at-a-glance
- Implicit state in motor PWM - forget to reset motor to full speed

With State Machines

Let's make things clearer by following the state machine model

Write the transition function

```
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 {
    if (line_pos != -1) {
      return FOUND;
    }
  }
}
```



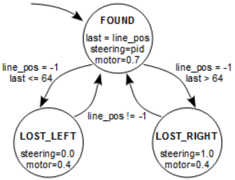
lost track state machine graphical notation

With State Machines

Let's make things clearer by following the state machine model

Write the state actions

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

void state_action(State state, int16_t line_pos, int16_t&
      last) {
  if (state == FOUND) {
    set_steering_pct(pid_update(line_pos));
    set_motor_pwm(0.7);
    last = line_pos;
  } else if (state == LOST_LEFT) {
    set_steering_pct(0.0);
    set_motor_pwm(0.4);
  } else if (state == LOST_RIGHT) {
    set_steering_pct(1.0);
    set_motor_pwm(0.4);
  }
}
```



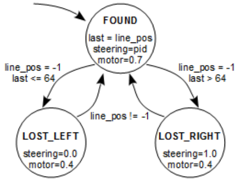
lost track state machine graphical notation

With State Machines

Let's make things clearer by following the state machine model

... and put it all together

```
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

Convenience vs. Performance

DigitalOutput

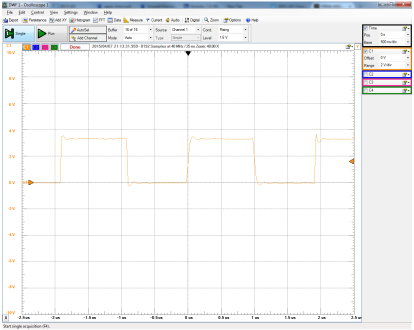
Given this simple block of code, guess the waveform frequency...

```
DigitalOut wave(PTB2);
while(1) {
  wave = !wave;
}
```

DigitalOutput

Given this simple block of code, guess the waveform frequency...

```
DigitalOut wave(PTB2);
while(1) {
  wave = !wave;
}
```



About 0.5MHz!
(or 1 edge per us)
That's at least an order of magnitude slower than the instruction clock!

Where might the bottleneck be?

Under the Hood: How DigitalOut Works

mbed/api/DigitalOut.h

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

mbed/targets/hal/TARGET_Freescale/TARGET_KLXX/gpio_object.h

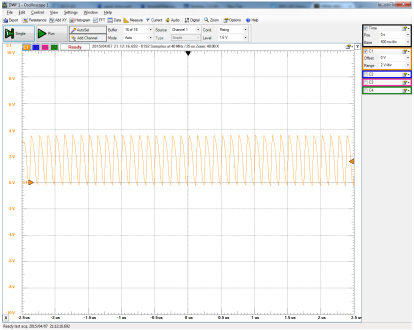
```
typedef struct {
    PinName  pin;
    uint32_t mask;
    __IO uint32_t *reg_dir;
    __IO uint32_t *reg_set;
    __IO uint32_t *reg_clr;
    __I  uint32_t *reg_in;
} gpio_t;

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

Many levels of indirection for a simple register write!

Raw register access

What if we skip the mbed API and directly write the register?

```
DigitalOut wave(PTB2); // set pin as output
while(1) {
  PTB->PTOR = (0x01 << 2); // set toggle register to flip pin PTB2
}
```



Much faster: about 8MHz!
(or 16 edges per us)

Each GPIO port has these registers:

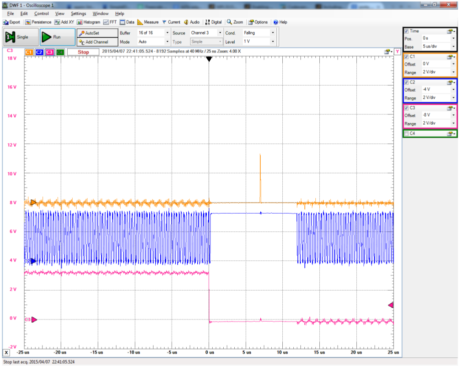
PDOR: set data
PSOR: set bits
PCOR: clear bits
PTOR: toggle bits
PDIR: input
PDDR: directionality

See MKL25Z4.h for details

InterruptIn Latency

Similarly, let's measure the InterruptIn latency

- ch1 (yellow) spike is ISR body
- ch2 (blue) toggling is main loop
- ch3 (pink) is interrupt signal
- Interrupts enabled using InterruptIn.fall(...)

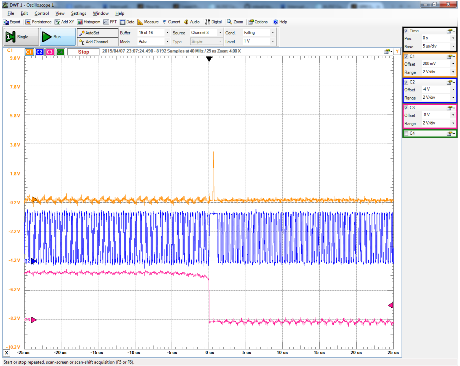


About 7us from edge to interrupt

InterruptIn Latency

What about a lower level implementation?

```
extern "C" void PORTA_IRQHandler() {
  PTB->PTOR = 0x04; PTB->PTOR = 0x04; // toggle ch1 (yellow)
  PORTA->ISFR = PORT_ISFR_ISF_MASK; // clear interrupt flags
}
NVIC_SetVector(PORTA_IRQn, (uint32_t)PORTA_IRQHandler); // set interrupt handler function
PORTA->PCR[16] = (PORTA->PCR[16] | PORT_PCR_IRQC_MASK); // enable on PTC16 / ch3 (pink)
NVIC_EnableIRQ(PORTA_IRQn);
```



Much faster: about 0.5us from edge to interrupt

But does this really matter?

- Order of magnitude faster
- ... but it's still microseconds
- Unlikely to be a bottleneck

Summary

- Interrupts and threading can make multitasking easier
  - Also come with their set of pitfalls and issues
- Write good code so you don't hate yourself later
- If you have high performance requirements, go below the mbed API
  - But in absolute timing terms, unlikely to make a significant difference

- Questions? Feedback?