EECS 192: Mechatronics Design Lab

Discussion 9 (Part 2): Embedded Software

written by: Richard "Ducky" Lin Spring 2015

18 & 19 Feb 2015 (Week 9)

- Embedded Programming
- Software Engineering

Embedded Programming

Hardware Specs

Recall the hardware specs for your boards:

- MKL25Z128VLK4 microcontroller
  - 48MHz ARM Cortex-M0+
  - 128KB flash
  - 16KB SRAM

What might make embedded programming different from desktop programming?

FRDM-KL25Z Board

image from KL25Z User's Manual

Memory Use

Say, I want to allocate some storage when I read my camera array.

```
uint16_t* read_camera() {
  uint16_t* camera_data = malloc(2*CAMERA_PIXELS);
  for (int i=0; i<CAMERA_PIXELS; i++) {
    camera_data[i] = camera_read_pixel();
  }
  return camera_data;
}
```

Why might this be a bad idea on a microcontroller?

Memory Use

Say, I want to allocate some storage when I read my camera array.

```
uint16_t* read_camera() {
  uint16_t* camera_data = malloc(2*CAMERA_PIXELS);
  for (int i=0; i<CAMERA_PIXELS; i++) {
    camera_data[i] = camera_read_pixel();
  }
  return camera_data;
}
```

Why might this be a bad idea on a microcontroller?

- Not checking for `malloc` failures - can return `NULL`
  - (this isn't an embedded-specific issue!)
- Dynamic (heap) memory allocation (`malloc`/`free`) is expensive
- Can cause heap fragmentation, especially when memory is scarce

Memory Use

Ok, so malloc is bad. I'm more an object-oriented C++ guy anyways!

```
CameraArray* read_camera() {
  CameraArray* camera_data = new CameraArray();
  camera_data->read_from(near_cam);
  return camera_data;
}

class CameraArray {
public:
  void read_from(Camera& camera);
  int8_t get_line_error();
protected:
  uint16_t camera_data[CAMERA_PIXELS];
}
```

Why is this also bad?

Memory Use

Ok, so malloc is bad. I'm more an object-oriented C++ guy anyways!

```
CameraArray* read_camera() {
  CameraArray* camera_data = new CameraArray();
  camera_data->read_from(near_cam);
  return camera_data;
}

class CameraArray {
public:
  void read_from(Camera& camera);
  int8_t get_line_error();
protected:
  uint16_t camera_data[CAMERA_PIXELS];
}
```

Why is this also bad?

- `new` also does dynamic memory allocation
  - So exactly the same issues as malloc, but perhaps a bit more sneaky

Pass-By-Value

Ok enough with dynamic memory allocation. No `new` either.

```
CameraArray read_camera(CameraArray camera_data) {
  camera_data.read_from(near_cam);
  return camera_data;
}

class CameraArray {
public:
  void read_from(Camera& camera);
  int8_t get_line_error();
protected:
  uint16_t camera_data[CAMERA_PIXELS];
}
```

What performance issues might arise from this?

Pass-By-Value

Ok enough with dynamic memory allocation. No `new` either.

```
CameraArray read_camera(CameraArray camera_data) {
  camera_data.read_from(near_cam);
  return camera_data;
}

class CameraArray {
public:
  void read_from(Camera& camera);
  int8_t get_line_error();
protected:
  uint16_t camera_data[CAMERA_PIXELS];
}
```

What performance issues might arise from this?

- C++ arguments are passed by value - it may create a copy
  - Copying large data structures is inefficient and can cause subtle bugs
- Pass pointers to objects or use references instead

Memory Use

Ok, let's say I write a recursive image processing algorithm.

Bear with me on this crappy example; I'm not a CV guy

```
uint8_t difference_gaussians(uint8_t level, uint16_t[] line_data) {
  uint16_t line_filtered[CAMERA_PIXELS];
  gaussian_blur(line_filtered /*dst*/, line_data /*src*/);
  if (level != 0) {
    uint8_t next_result = difference_gaussians(level-1,
        line_filtered);
  }
  return /*CV magic on filtered and original line data*/;
}
```

So what can go wrong here?

Memory Use

Ok, let's say I write a recursive image processing algorithm.

Bear with me on this crappy example; I'm not a CV guy

```
uint8_t difference_gaussians(uint8_t level, uint16_t[] line_data) {
  uint16_t line_filtered[CAMERA_PIXELS];
  gaussian_blur(line_filtered /*dst*/, line_data /*src*/);
  if (level != 0) {
    uint8_t next_result = difference_gaussians(level-1,
        line_filtered);
  }
  return /*CV magic on filtered and original line data*/;
}
```

So what can go wrong here?

- Potential stack overflow if recursion runs deep enough
  - Each recursive call allocates a 2*CAMERA_PIXELS array on stack
  - Possibly undetected (no operating system or memory protection)!

Synchronization

Ok, let's talk threads!

```
uint16_t camera_data[CAMERA_PIXELS];
void camera_read_thread() {
  for (int i=0; i<CAMERA_PIXELS; i++) {
    camera_data[i] = camera_read_pixel();
  }
  Thread.wait(INTEGRATION_TIME);
}
void camera_process_thread() {
  uint8_t line_camera_distance = /*magic filter*/;
  servo_pwm.write(kp * line_camera_distance);
}
```

What might happen?

Synchronization

Ok, let's talk threads!

```
uint16_t camera_data[CAMERA_PIXELS];
void camera_read_thread() {
  for (int i=0; i<CAMERA_PIXELS; i++) {
    camera_data[i] = camera_read_pixel();
  }
  Thread.wait(INTEGRATION_TIME);
}
void camera_process_thread() {
  uint8_t line_camera_distance = /*magic filter*/;
  servo_pwm.write(kp * line_camera_distance);
}
```

What might happen?

- No synchronization! Can read data in the middle of a write!
  - Might get half of one frame and half of another...

Synchronization

Ok, let's talk threads!

```
uint16_t camera_data[CAMERA_PIXELS];
void camera_read_thread() {
  for (int i=0; i<CAMERA_PIXELS; i++) {
    camera_data[i] = camera_read_pixel();
  }
  Thread.wait(INTEGRATION_TIME);
}
void camera_process_thread() {
  uint8_t line_camera_distance = /*magic filter*/;
  servo_pwm.write(kp * line_camera_distance);
}
```

How do I prevent it?

Synchronization

Ok, let's talk threads!

```
uint16_t camera_data[CAMERA_PIXELS];
void camera_read_thread() {
  for (int i=0; i<CAMERA_PIXELS; i++) {
    camera_data[i] = camera_read_pixel();
  }
  Thread.wait(INTEGRATION_TIME);
}
void camera_process_thread() {
  uint8_t line_camera_distance = /*magic filter*/;
  servo_pwm.write(kp * line_camera_distance);
}
```

How do I prevent it?

- Various synchronization constructs: mutexes/locks, semaphores, ...
- Nonblocking solutions: double/triple buffering
  - Or asynchronous FIFOs (efficiently implemented as a circular buffer)

Software Engineering

Two Cameras

I have some code to read a single camera.

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

void control_loop() {
  servo_pwm.write(kp * near_cam.get_line_distance());
}
```

Given the structure, how would I add another camera?

Two Cameras

I have some code to read a single camera.

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

void control_loop() {
  servo_pwm.write(kp * near_cam.get_line_distance());
}
```

Given the structure, how would I add another camera?

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

Two Cameras

I have some code to read a single camera.

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

void control_loop() {
  servo_pwm.write(kp * near_cam.get_line_distance());
}
```

Given the structure, how would I add another camera?

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

What hidden assumptions / expectations did I have for `Camera`?

Expectations

What if the Camera implementation looked like this?

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

class Camera {
public:
  Camera(PinName clk, PinName si, PinName adc);
  void read() {
    /*ADC reads into global camera_data*/
  }
  int8_t get_line_distance() {
    return /*some computation on global camera_data*/;
  }
}
```

Expectations

What if the Camera implementation looked like this?

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

class Camera {
public:
  Camera(PinName clk, PinName si, PinName adc);
  void read() {
    /*ADC reads into global camera_data*/
  }
  int8_t get_line_distance() {
    return /*some computation on global camera_data*/;
  }
}
```

OH SH-

- Breaks user expectations of object encapsulation and independence
  - DON'T DO IT!

Globals

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

```
float motor_velocity_target; //global

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

So far, so good, right?

Globals

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

```
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;
```

Globals

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

```
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?

```
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;`

Soon, you have no clue what the target actually is - dataflow spaghetti!

Summary

- Avoid dynamic memory allocation
- Watch out for the limited RAM and stack overflow
- Watch out for synchronization errors
- Write code that conforms to user expectations
- Avoid dataflow spaghetti