I. Reentrancy and Atomic Operations

- 3 rules:
  - Use shared variables in an atomic way
  - Don’t call non-reentrant functions
  - Don’t use hardware in a non-atomic way

(temp = foobar; temp += 1; foobar = temp;)

OR

foobar+=1;

What does the compiler do?
Compiler output:

(x86 compiler)
moveax,[foobar]
incax
mov[foobar],ax

Atomic version:
inc[foobar]

Moral: Don't trust your compiler!

Automatic variables

int foo;
void some_function(void) {
  foo++;
}

void some_function(void) {
  int foo;
  foo++;
}

Doesn't work! if called from code with interrupts disabled…

Keeping Code Reentrant

long I;
void do_something(void) {
  disable_interrufts();
  i+=0x1234;
  enable_interrufts();
}

Doesn't work! if called from code with interrupts disabled…

Better:

long I;
void do_something(void) {
  push interrupt_state;
  disable_interrufts();
  i+=0x1234;
  enable_interrufts();
  pop interrupt_state;
}

Or, use semaphores or RTOS locking mechanism
Hardware reentrancy

```c
int timer_hi;
interrupt timer()
{
    ++timer_hi;
}
```

```c
long read_timer(void)
{
    unsigned int low, high;
    low = inword(hardware_register);
    high = timer_hi;
    return (high<<16 + low);
}
```

This code will fail, occasionally...

One failure mode:

1. `read_timer` reads the hardware and gets 0xffff
2. Immediately the timer hardware increments to 0x0000
3. The overflow triggers an interrupt. The ISR runs, and increments `timer_hi` to 0x0001, not 0x0000 as in step 1
4. The ISR returns, our `read_timer` concatenates the new 0x0001 with the previously read 0xffff, and returns 0x1ffff – WRONG!!!

Or, while interrupts are disabled:

1. `read_timer` starts. The timer is 0xffff with no overflows.
2. Before much else happens it increments to 0x0000. With interrupts off the pending interrupt gets deferred.
3. `read_timer` returns a value of 0x0000 instead of the correct 0x10000, or the reasonable 0xffff.

A once-a-month bug? How do you find it?

Solutions:

- Stop the timer BEFORE reading!
  Downside: we lose time.
- Or, read `timer_hi`, then the hardware timer, then re-read `timer_hi`. Iterate until the two variable reads are equal.
  Downside: can take a long time in a heavily loaded system
II. Pulse Width Modulation

Why pulse-width modulation works

- Most mechanical systems are low-pass filters
  - Consider frequency components of pulse-width modulated signal
  - Low frequency components affect components
    - They pass through
  - High frequency components are too fast to fight inertia
    - They are "filtered out"
- Electrical RC-networks are low-pass filters
  - Time constant (τ = RC) sets "cutoff" frequency
    that separates low and high frequencies

Pulse-width modulation

- Pulse a digital signal to get an average “analog” value
- The longer the pulse width, the higher the voltage

Pulse-width ratio = \( \frac{t_{on}}{t_{period}} \)

Most mechanical systems are low-pass filters

Anti-lock brake system

- Normal operation
  - Regulate velocity of rear wheel
- Brake pressed
  - Gradually increase amount of breaking
  - If skidding (front wheel is moving much faster than rear wheel) then temporarily reduce amount of breaking

Inputs

- Brake pedal
- Front wheel speed
- Rear wheel speed

Outputs

- Pulse-width modulation rear wheel velocity
- Pulse-width modulation brake on/off
**Rear wheel controller/anti-lock brake system**

- brake pedal pressed
- front wheel velocity
- rear wheel velocity
- micro controller
- brake on/off
- move rear wheel

**Basic I/O ports (brakes)**

- Check if brake pedal pressed – or interrupt
  - brakePressed = read (brakePedalPort)
- Turn brake on/off
  - write (brakePort, onOff)
- Move rear wheel
  - write (rearWheel, onOff)

**Polling vs. interrupts**

- Software must repeatedly check
  - Brake pedal port
  - How often?
  - Need to make sure not to forget to do so (use timer)
- Use automatic detection capability of processor
  - Connect brake pedal to input capture or external interrupt pin
  - Interrupt on level change
  - Interrupt handler for brake pedal

**Pulse-width modulation for brakes**

- To pump the brakes gradually increase the duty-cycle \( t_{on} \) until car stops
Brake pump setup

- Use timer to turn brake on and off
  - Apply brake
  - Set timer to interrupt after "on" time
  - Disengage brake
  - Set time to interrupt after "off" time
  - Repeat
- How do we tell which interrupt is which?

**Diagram:**

```
set timer to go off at each edge
```

Brake pump setup (cont’d)

- Change value of "on" time to change analog average
  - average output = ( on + off ) / period
- How do we decide on the period of the pulses?
- Using two timers
  - One to set period (auto-reload)
  - One to turn it off at the right duty cycle

**Diagram:**

```
set timer to go off at each edge
```

Bright LED

- Easy to control intensity of light through pulse-width modulation
- Duty-cycle is averaged by human eye
  - Light is really turning on and off each period
  - Too quickly for human retina (or most video cameras)
  - Period must be short enough (< 1ms is a sure bet)
- LED output is low to turn on light, high to turn it off
  - Active low output

**Code snippet:**

```c
volatile uint8_t width; /* positive pulse width */
volatile uint8_t delay; /* used to slow the pulse width changing */

SIGNAL (SIG_OVERFLOW2)
{
  if(delay++ == 25) { OCR2 = width++; delay = 0; }
}

int main (void)
{
  /* must make OC2 pin an output for the PWM to visible */
  DDOR = _BV(DDD7);
  /* use Timer 2 FastPWM and the overflow interrupt to update duty-cycle */
  TCCR2 = _BV(WGM21) | _BV(WGM20) | _BV(COM21) | _BV(COM20) | _BV(CS21) | _BV(CS20);
  TIMSK = _BV(TOIE2);
  /* setup initial conditions */
  delay = 0;
  /* disable interrupts */
  sei ();
  for (i = 0; i < 1000; i++) { 
    /* LOOP FOREVER as the interrupt will make necessary adjustment */
  } return (0);
}
```

Sample code for LED

- Varying PWM output
Control three LEDs: Red, Green, and Blue
- Active Low, 15 ms period, 256 possible values for LED
- Timer 0 interrupt 15ms/256
- Each interrupt, dec. count and decide if each LED is on
  - if count is 0, count is 255 and all LEDs are off

III. Color

Color perception usually involves three quantities:
- **Hue**: Distinguishes between colors like red, green, blue, etc
- **Saturation**: How far the color is from a gray of equal intensity
- **Lightness**: The perceived intensity of a reflecting object

Sometimes lightness is called *brightness* if the object is emitting light instead of reflecting it.

In order to use color precisely in computer graphics, we need to be able to specify and measure colors.
**Color Spaces**

- **Definition:** A mapping of color components onto a Cartesian coordinate system in three or more dimensions.
- **Examples:** RGB, CMY, XYZ, HSV, HLS, Lab, UVW, YUV, YCrCb, Luv, L* u* v*, ...
- **Different Purposes:** display, editing, computation, compression, ...
- **Equally distant colors may not be equally perceivable**

**Additive Model:**

**RGB System**

- R, G, B normalized on orthogonal axes
- All representable colors inside the unit cube
- Color Monitors mix R, G and B
- Video cameras pick up R, G and B
- 3 fixed components acting alone can’t generate all spectrum colors.

**RGB Color space**

**Problems with RGB**

- Only a small range of potential perceivable colors (particularly for monitor RGB)
- It isn’t easy for humans to say how much of RGB to use to get a given color
  - How much R, G and B is there in “brown”?
- Perceptually non-linear
  - Two points, a certain distance apart, may be perceptually different in one part of the space, but could be same in another part of the space.
Subtractive model (CMY System)
- Color results from removal of light from the illumination source
- Pigments absorb R, G or B and so give C, M or Y
- Used in deskjet/inkjet printers.
- No ink (pigment) = white

Converting between RGB and CMY

Specifying Color
- Color perception usually involves three quantities:
  - Hue: Distinguishes between colors like red, green, blue, etc
  - Saturation: How far the color is from a gray of equal intensity
  - Lightness: The perceived intensity of a reflecting object

Sometimes lightness is called brightness if the object is emitting light instead of reflecting it.
How Do Artists Do It?

- Artists often specify color as tints, shades, and tones of saturated (pure) pigments

- **Tint**: Gotten by adding white to a pure pigment, decreasing saturation

- **Shade**: Gotten by adding black to a pure pigment, decreasing lightness

- **Tone**: Gotten by adding white and black to a pure pigment

---

HSV Color Space

- Computer scientists frequently use an intuitive color space that corresponds to tint, shade, and tone:

  - **Hue** - The color we see (red, green, purple)
  - **Saturation** - How far is the color from gray (pink is less saturated than red, sky blue is less saturated than royal blue)
  - **Brightness (Luminance)** - How bright is the color (how bright are the lights illuminating the object?)

---

HSV Color Model

- **Hue (H)** is the angle around the vertical axis

- **Saturation (S)** is a value from 0 to 1 indicating how far from the vertical axis the color lies

- **Value (V)** is the height of the hexcone
**HSV Color Space**

- A more intuitive color space
  - H = Hue
  - S = Saturation
  - V = Value (or brightness)

[Image of HSV color space]

[Link to: http://www.cs.rit.edu/~ncs/color/a_spaces.html]

**HSV System**

- Normally represented as a cone or hexcone
- Hue is the angle around the circle or the regular hexagon; $0 \leq H \leq 360$
- Saturation is the distance from the center; $0 \leq S \leq 1$
- Value is the position along the axis of the cone or hexcone; $0 \leq V \leq 1$
- Value is not perceptually-based, so colors of the same value may have slightly different brightness
- Main axis is grey scale

```java
if ( S == 0 )                       //HSV values = From 0 to 1
{ R = V * 255                      //RGB results = From 0 to 255
  G = V * 255
  B = V * 255
} else
{
  var_h = H * 6
  var_i = int( var_h )             //Or ... var_i = floor( var_h )
  var_1 = V * (1 - S)
  var_2 = V * (1 - S * (var_h - var_i))
  var_3 = V * (1 - S * (1 - (var_h - var_i)))

  if ( var_i == 0 ) { var_r = V ; var_g = var_3 ; var_b = var_1 }
  else if ( var_i == 1 ) { var_r = var_2 ; var_g = V ; var_b = var_1 }
  else if ( var_i == 2 ) { var_r = var_1 ; var_g = V ; var_b = var_3 }
  else if ( var_i == 3 ) { var_r = var_1 ; var_g = var_2 ; var_b = V }
  else                          { var_r = V ; var_g = var_1 ; var_b = var_2 }

  R = var_r * 255                  //RGB results = From 0 to 255
  G = var_g * 255
  B = var_b * 255
}
```