Final Exam Cheat Sheet 2

Register's vs. Variables

Registers

What are they?:
- Suggestion to the compiler to use register to store variable, but NOT guranteed by compiler.
- Declared: **register int** i = 10;

Benefits:
- Register's are faster.

Scope: N/A

Local Variables

What are they?: Variables defined and used within a specific function and have no meaning beyond that function.

Benefits:
- Help Compiler Optimize:
  - Limit the scope of each variable.
  - Within each scope, limit the number of variables.
  - Better chances for compiler to free up registers.

Scope:
- Within function it's defined.
- Example Below:

```
for (i=0;  i < MAX_ARRAY_LENGTH; i++) { array[i] = i; }
_ /* do stuff to array, need to set it up again */
/* i still equals max array so subtract one and run through the loop again*/
for (i--; i >= 0; i--) { array[i] = i; }

for (i=0;  i < MAX_ARRAY_LENGTH; i++) { array[i] = i; }
_ /* do stuff to array, need to set it up again */
for (i=0;  i < MAX_ARRAY_LENGTH; i++) { array[i] = i; }
```

Global Variables

What are they?: Global variables are defined and used within multiple functions and have meaning in the entire program.

Benefits:
- Pros
  - Share variables in function calls (Don't pass as parameter).
- Cons
  - Outside the flow of code.
  - Reentrant functions.
  - Cannot be stored in registers.

Scope:
- Entire program.

Function Reduction

Concept: Function reduction is a form of optimization to reduce small repetative functions or operations to reduce the total number of iterations. Examples below:

Combine Functions

Before

```
void LcdWriteBuffer(uint16_t* buffer, uint16_t buflength){

    int i;
    while (buflength){

        LcdWriteBuffer(buffer[i] & 0xFF);     // Write lower byte
        LcdWriteBuffer(buffer[i] .. 8) & 0XFF);// Write upper byte
        i++
        buflength--;
    }
}
void LcdWriteBusByte(uint8_t data){
    IoClear(LCD_SELECT_N);                    // Select the chip
    IoWriteBusByte(LCD_BUS, data);  // Write the I/O lines
    IoSet(LCD_SELECT_N);                      // Deselect the chip
}
IoWriteBusByte(uint32_t io, uint8_t data){
    // ioBus was configured during initialization
    ioBus[io] = data;
}
```

After

```
void LcdWriteBuffer(uint16_t* buffer, uint16_t buflength){

    int i;
    IoClear(LCD_SELECT_N);                    // Select the chip
    while (buflength){
        LcdWriteBuffer(buffer[i] & 0xFF);     // Write lower byte
        LcdWriteBuffer(buffer[i] .. 8) & 0XFF); // Write upper byte
        i++
        buflength--;
    }
    IoSet(LCD_SELECT_N);                      // Deselect the chip
}
```

Change Iterators

T1: (add E1) E1 is added so it becomes the currEvent. Everything else is END

T2: (add E0) E1 is still the currEvent... and E0 goes in [1: x] (because last item added was E1)

T3: (add E3) E1 is still the currEvent... and E3 goes in [0: x] (because last item added was E0)

T4: (pop E1) E0 is now the currEvent, bc we popped E1. and [1: END]

T5: (add E2) E0 is still the currEvent... and E2 goes in [3: x] (because the last item added was E3)

Reduce Math

Good

```
IoWriteBusByte(LCD_BUS, *buffer & 0xFF); // Write lower buffer

    Variable i would already be in a regite, already initalized.
    Read the contents of memory at that address
    Perform bitwise AND with contents
    Put buffer on stack
    Call IoWriteBusByte, passing data
    Pop buffer off the stack
}

IoWriteBusByte(LCD_BUS, (buffer[i]>>8) & 0xFF){ // Write upper buffer

    Variable i would already be in a regite, already initalized.
    Read the contents of memory at that address
    Perform shift with contents
    Perform bitwise AND with result
    Put buffer on stack
    Call IoWriteBusByte, passing data
    Pop buffer off the stack
}
buffer++; // Increment buffer
```

Choose a proper type / size pointer.

```
void LcdWriteBuffer(uint16_t* buffer, uint16_t buflength){
    uint8_t &byteBuffer = (uint8 *) buffer;
    // Buffer is now a buffer of bytes.
    // This works ONLY if your endian-ness match
    // what the hardware expects.
    buflength = buflength*2;

    IoClear(LCD_SELECT_N);                    // Select the chip
    while(buflength){
        IoWriteBusByte(LCD_BUS, *Bufferbyte);
        buflength--;
        byteBuffer++;
    }
    IoSet(LCD_SELECT_N);                      // Deselect the chip
}
```

Unroll Loops

Before

```
void foo(uint8_t* dst, uint8_t* src, uint8_t len) {
    while (len--) {
        dst[len] = src[len];
    }
}
```

After

```
void bar(uint8_t* dst, uint8_t* src, uint8_t len) {
    while (len--) {
        dst[len] = src[len];
        len--;
        dst[len] = src[len];
    }
}
```

Example | Solving Fmax, Fmin, and Energy Consumption (HW6)

(a) An application has three events: E1, E2 and E3. The application runs in an embedded system with a 40MHz clock. E1 occurs every 100ms and needs 20ms for execution. E2 occurs every 30ms and needs 3ms for execution. E3 occurs every 40ms and needs 3ms for execution.

a.1) Over 3 seconds, how many times do E1, E2 and E3 occur respectively, and how much execution time do E1, E2 and E3 need respectively? What we know:

CPU: $40MHz$

Event 1: $C1 = 20ms$, $P1 = 100ms$ **Event 2:** $C2 = 3ms$, $P2 = 30ms$ **Event 3:** $C3 = 3ms$, $P3 = 40ms$

Answer: a.1)

∴ First thing we need to do is covert all period's for E1, E2, and E3 to "occurences per second" instead of "occurences per millisecond":

Event 1:
$$100ms * \frac{1sec}{1000ms} = 0.1sec$$

Event 2:
$$30ms * \frac{1sec}{1000ms} = 0.03sec$$

Event 3:
$$40ms * \frac{1sec}{1000ms} = 0.04sec$$

∴ Next we need to determine how long each event occures in the given period of 3 sec:

Event 1:
$$\frac{3sec}{0.1sec} = 30times$$

Event 2:
$$\frac{3sec}{0.03sec} = 100times$$

Event 3:
$$\frac{3sec}{0.04sec} = 75times$$

∴ Next we will convert execution times to "execution per second" instead of "execution per millisecond":

Event 1:
$$20ms * \frac{1sec}{1000ms} = 0.02sec$$

Event 2:
$$3ms * \frac{1sec}{1000ms} = 0.003sec$$

Event 3:
$$3ms * \frac{1sec}{1000ms} = 0.003sec$$

∴ Next we will solve for the total execution time for each event in the given period of 3sec:

Event 1: $30times * 0.02sec = 0.6sec$ ***OR*** $0.6sec * \frac{1000ms}{1sec} = 600ms$

Event 2: $100times * 0.003sec = 0.300sec$ OR $0.300sec * \frac{1000ms}{1sec} = 300ms$

Event 3: $75times * 0.003sec = 0.225sec$ OR $0.225sec * \frac{1000ms}{1sec} = 225ms$

Total Execution Time: $0.6sec + 0.300sec + 0.225sec = 1.125sec$

a.2) Can E1, E2 and E3 be scheduled in this application?

Answer: a.2)

∴ First we need to determine the $Sum \frac{Ci}{Pi}$ of E1,E2, and E3:

Event 1: C1 = 20ms, P1 = 100ms
$$Sum\frac{C1}{P1} = \frac{20ms}{100ms} = 0.2$$

Event 2: C2 = 3ms, P2 = 30ms
$$Sum\frac{C2}{P2} = \frac{3ms}{30ms} = 0.1$$

Event 3: C3 = 3ms, P3 = 40ms
$$Sum\frac{C3}{P3} = \frac{3ms}{40ms} = 0.075$$

$$Sum\frac{Ci}{Pi} = \frac{20ms}{100ms} + \frac{3ms}{30ms} + \frac{3ms}{40ms}$$

$$Sum\frac{Ci}{Pi} = 0.375$$

∴ We need to plug our calculation into the forumla: $Sum\frac{Ci}{Pi} < 1$, which lends to $0.375 < 1$

since the condition of $Sum\frac{Ci}{Pi} < 1$ is satisified then **we can schedule E1, E2 and E3 in this application**.

(b) Power is proportional to frequency, i.e. P=cF. Assume the operating power at 40MHz is 40mW and the idle power at any frequency is 0.1mW.

b.1) To reduce power, the application wants to lower the clock frequency. The three events will occur at the same rate, but their execution times will be enlarged. What is the minimum frequency that the three events can be scheduled?

What we know:

$Fmax: 40MHz$

Event 1: C1 = 20ms, P1 = 100ms
Event 2: C2 = 3ms, P2 = 30ms
Event 3: C3 = 3ms, P3 = 40ms

$$Fmin = Fmax * Sum\frac{Ci}{Pi}$$

$$Sum\frac{Ci}{Pi} = \frac{C1}{P1} + \frac{C2}{P2} + \frac{C3}{P3}$$

Answer: b.1)

∴ We know from question 1.a.1 that the three events E1, E2, and E3 are schedulable in the timeframe of 3 seconds. Thus to solve for $Fmin$ we simply need to plug the values found in 1.a into $Fmin = Fmax * Sum\frac{Ci}{Pi}$.

$$\therefore Sum\frac{Ci}{Pi} = \frac{20ms}{100ms} + \frac{3ms}{30ms} + \frac{3ms}{40ms}$$

$$\therefore Sum\frac{Ci}{Pi} = 0.375$$

$$\therefore Fmin = Fmax * Sum\frac{Ci}{Pi}$$

$$\therefore Fmin = 40Mhz * 0.375$$

$$\therefore Fmin = 40Mhz * 0.375$$

$$\therefore Fmin = 15Mhz$$

b.2) Over 3 seconds, how much energy is consumed at 40MHz.

What we know:

$Fmax: 40Mhz$ $\quad idleTime = ???$
$OperatingPower = 40mW$ $\quad P = cF$
$OperatingTime = 1.125sec$ $\quad Energy = EnergyActive * EnergyIdle$
$IdlePower = 0.1mW$

Answer: b.2)

∴ We must find the total *Idle Time*:

$$TotalIdleTime = TotalTime - TotalExecutionTime$$
$$TotalIdleTime = 3sec - 1.125sec$$
$$TotalIdleTime = 1.875sec$$

∴We can use the equation $Energy = Power * Time$ to determing how much energy is consumed by 40MHz processor. We must account for the *Active Energy* and *Idle Energy*, so we will modify the equation to:

$$\therefore Energy = EnergyActive * EnergyIdle$$

$$\therefore EnergyActive = PowerActive * TimeActive$$

$$= 40mW * 1.125sec$$

$$\therefore EnergyIdle = PowerIdle * TimeIdle$$

$$= 0.1mW * 1.875sec$$

$$\therefore Energy = ((40mW * \frac{1W}{1000mW}) * 1.125sec) + ((0.1mW * \frac{1W}{1000mW}) * 1.875sec)$$

$$\therefore Energy = 0.0045Joules + 0.001875Joules$$

$$\therefore Energy = 0.0451875Joules$$

Example | Solving Fmax, Fmin, and Energy Consumption (HW6) [Continued]

b.3) Over 3 seconds, how much energy is consumed at the minimum frequency? **(found in b.1)**

What we know:

$Fmin: 15MHz$ $\quad IdlePower = 0.1mW$
$OperatingPower = 40mW$ $\quad IdleTime = 1.875sec$
$OperatingTime = 1.125sec$ $\quad P = cF$

Answer: b.3)

∴ We can use the equation to determing how much power is consumed at our minimum frequency of from b.1.

$$MinimumPower : P = cF$$

∴ We need to solve for "c" from $P = cF$ to use it to find the *Minimum Power*. We can use the "P" and "F" from 1.b.2 to solve for "c":

(1.b.2) $P = 40mW$ (1.b.2) $F = 40MHz$

$$c = \frac{P}{F}$$

$$c = \frac{40mW}{40MHz}$$

$$c = 1$$

∴ Now we can solve for $MinimumPower : P = cF$:

$$MinimumPower : P = cF$$
$$P = 1 * 15Mhz$$
$$P = 15mW$$

∴ We can now use the equation $Energy = Power * Time$ to determing how much energy is consumed at our minimum frequency of $15MHz$ from 1.b.1. We must account for the *Active Energy* and *Idle Energy*, so we will modify the equation to:

$$\therefore Energy = EnergyActive * EnergyIdle$$

$$\therefore EnergyActive = PowerActive * TimeActive$$

$$= 15mW * 1.125sec$$

$$\therefore EnergyIdle = PowerIdle * TimeIdle$$

$$= 0.1mW * 1.875sec$$

$$\therefore Energy = ((15mW * \frac{1W}{1000mW}) * 1.125sec) + ((0.1mW * \frac{1W}{1000mW}) * 1.875sec)$$

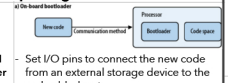
$$\therefore Energy = 0.016875Joules + 0.001875Joules$$

$$\therefore Energy = 0.017063Joules$$

Updating Code

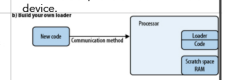
Ways to Update

On-board Bootloader



- Set I/O pins to connect the new code from an external storage device to the embedded system.
- Load code from external storage device to the *Code Space* on the embedded device.

BYOL Build Your Own Loader



Update from RAM

Key Components
- New code storage (EEPROM Volumes)
- Code transmission (USB / Network)
- Code Integrity Check (CRC)
- Scratch *Code Space:*
  - Data RAM.
  - Another EEPROM volume.
- Non-distruption controller:
  - Bootloader in AUX flash

Main Issue with Updating

The main issue with updating the code on an embedded device is access to *Program Memory*. Access is restricted and the update process is fragile and prone to errors including:
- **Code block damage** (prior to updating).
- **Flash memory rewrite errors** (during update).
- **Power loss** (during update).

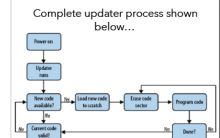
The aforementioned errors can lead to a *Bricked* device if not properly handled and recovered from using previous versioned code.

BYOU Build Your Own Updater

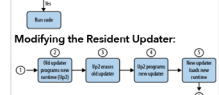
Overview:
A resident program in the *Code Space* that can reprogram the embedded devices memory on it's own.

Update Process:
1. Erase a sector (always MCU specific).
2. Read the new code into *Scratch Memory*.
3. Use chip-specific functions to write the code to the erased sector from 1.
4. Repeat *Steps 1.-3.* until ALL runtime sectors are completely overwritten.

Complete updater process shown below...



Modifying the Resident Updater:



Issues:
- New code is corrupted from the source or packets of the code are dropped in communication thus rendering the new code incomplete.
- New code storage or writing failures (power-loss).

Accuracy vs. Precision

Accuracy:
- How correct is the value.
- Accuracy is limited.

Precision:
- How many significant figures.
- Precision could be noise.

Speedy Operations

- Bit operations (<< >>)
- Addition (+)

```
for (i=0; i<100; i++) {
  if ((i%10) == 0) {
    printf("%d percent done.", i);
  }
}
```

BAD

GOOD

```
for (i=0; i<100; i++) {
  if (!(i & 0x07)) { // this will print out every 8th pass
    printf("%d percent done.", i);
```

Average

Average with a Rolling Window:

newAverage = lastAverage + (newSample/length) - (oldSample/length);

Problems:
- Overflow:
  - Addition of 2^n n-bit numbers.
  - (n+m)-bit result.
  - Integer multiplication.
- Underflow (Division Precision):
  - Integer division.
  - n-bit number divided by 2^n
  - (n-m)-bit result.

newAverage = lastAverage + ((newSample -oldestSample)/length);
- If the difference is smaller than the length, the new average will be the same.

Proper Average:
- Large values should be divided by much smaller ones.
- They shouldn't be similar in magnitude.
- Choose a proper length of window:
  - Addition can cause *overflow*.
  - Division can cause *underflow*.
- Scale number but choose a proper scale.
  - Multiplication can cause *overflow*.
- Know and depend upon a range of input values so that you can make sure your accuracy is not degraded by the lack of precision.

Divide a Constant

Formulas

Multiplier = Divisor/ (Number to Divide)
Divisor = 2^n
n = Shift Amount
Result = Multiplier/Divisor
%Error = |((desired-actual)/(desired))*100 |

Example

Multiplier	Divisor	Equivalent shift	Result	% Error
1	6	None	0.166666667	0
3	16	4	0.1875	12.5
5	23	5	0.15625	6.2
11	64	6	0.171875	3.1
21	128	7	0.164063	1.5
43	256	8	0.167969	0.78
85	512	9	0.166016	0.39
171	1024	10	0.166992	0.39
341	2048	11	0.166504	0.09
683	4096	12	0.166748	0.04

```
while(1){                                   if (event
  pollButton();                               turn of
  switch(st){                                 call ne
    case S0_Released:                           break;
      ledOff(1);
      if(evButton == Pressed){
        startDelay();
        st = S1_Pressed;
      }
      break;
    case S1_Presssed:
      ledOn(1);
      if(evTimer == Expired){
        st = s2_Pressed;
      }
      break;
    case S2_Pressed:
      if(evButton == Released){
        ledOff(1);
        startDelay();
        st = S3_Released;
      }
      break;
    case S3_Released:
      ledOff(1);
      if(evTimer == Expired){
        st = s0_Released;
      }
  }
}
```

```
bool isEmpty(Queue* que) {
  return que->ourEvent == END;
}
uint8_t pop(Queue* que) {
  _builtin_disi(0x3fff);
  uint8_t ev = que->ourEvent;
  que->ourEvent = que->q[ev];
  que->q[ev] = END;
  _builtin_disi(0x0000);
  return ev;
}
void append(Queue* que, uint8_t ev) {
  _builtin_disi(0x3fff);
  if(ev < QLEN){
    uint8_t curEv = que->ourEvent;
    if(curEv == END) que->ourEvent = ev;
    else{
      uint8_t qcnt = 0;
      while (curEv != END && qcnt < QLEN){
        if(curEv == ev) break;
        if(que->q[curEv] == END){
          que->q[curEv] = ev;
          que->q[ev] = END; break;
        }
        curEv = que->q[curEv]; qcnt++;
      }
    }
  }
  _builtin_disi(0x0000);
}
```

Main — Write take

Race to see which write wins

```
void initQueue(Queue* que) {
  que->ourEvent = END;
  uint8_t i = 0;
  for(i = 0; i < QLEN; i++)
    que->q[i] = END;
}
```

Race Conditions

Race Condition Definition

- Tasks writing to share memory at the same time causes unexpected results.
- ANY shared memory between tasks leads to inconsistent behavior.
- Race Conditions can be triggered by calling *non-reentrant functions*.

Solve Race Conditions

Mutual Exclusion:
- Critical resources are protected so only one function can modify them at a time.
- Mutex reuires tasks that CANNOT interrupted.

Atomic:
- Atomic is an action that CANNOT be interupted.
- Interrupts disabled while accessing protected resource.
- Interrupt latency increased / interrupts missed.

Scheduler: (See *Scheduler*)

Potential Lockup:

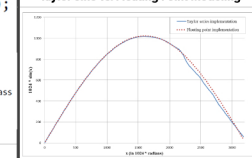
Finite State Machine (FSM):
- Separate *app variables* from *interrupt variables*:
- App Variables == States
- Interrupt Variables == Events
- App only READS interrupt variables.
- Interrupts do NOT read or write app variables.

Scaling Input/Output

Sine
- Scale **x** by 1024.
- All polynomial factors need to be scaled too.
- 1/3! * 1024 = 171

```
xSq = (x*x) >> 10;  // right-shift by 10 is equal to
tmp = - DivideSevenFactorial(xSq);
tmp = (xSq * (INVERSE_FIVE_FACTORIAL + tmp)) >> 10;
tmp = (xSq * (-INVERSE_THREE_FACTORIAL + tmp)) >> 10;
sinX = x - ((x * tmp)>>10);
```

Taylor Sine vs. Floating Point Modeling



Notice how Taylor Sine closely matches the the range of values for floating point arithmetic thus proving Taylor Sine is an accurate method for modeling floating pint arithmetic.

Lookup Tables

Concept:
Trade *Code Space* for speed. Common values for scaling are kept in a table so they can be looked up quickly later when needed.

Overview:
- Scale of Input: 1000x
- Scale of Output: 1000x (1024x)
- Step-size: Determines the number entries
  - 200.
  - Some power of 2.
  - Variable in size.
- Position of entries: Left/Center

Linear Interpolations:
- Way of modeling points along a curve or line.
- Uniform step-size NOT always the right approach:
  - Smaller steps on curves.
  - Larger steps on straight lines.

Explicit Loopup Table (ELT):
- ELT accounts for BOTH input and output values on the same lookup table.
- Coding Example: (See Below)

```
int SeachLookupTable(int32_t target, struct sPoint const *table, int tableSize) {
  int i;
  int bestIndex = 0;

  for (i=0; i<tableSize; i++) {
    if (target > table[i].x) {
      bestIndex = i;
    } else {
      return bestIndex;
    }
  }
  return bestIndex;
}

index = SeachLookupTable (x, sinLookup, sizeof(sinLookup));
if (index+1 < sizeof(sinLookup)) {
  y = interpolate(x, sinLookup[index], sinLookup [index + 1]);
} else {
  y = interpolate(x, sinLookup[index-1], sinLookup [index]);
}
```

Binary Scaling (Fake Floating Point)

Concept: Using a struct with binary shifting to mimick a floating point value.

Code Example:

```
struct sFakeFloat {
  int32_t num;  // numerator
  int8_t shift; // right-shift values (use negative for left-shift)
}

floatingPointValue = num >> shift;    // In the actual code

struct sFakeFloat oneFourth = {1, 2};

struct sFakeFloat four = {1, -2};
```

Precision:

Numerator	Number of bits needed in numerator	Denominator shift values	Equivalent floating point number	Error
12	4	0	12	0.345
25	5	1	12.5	0.355
99	7	3	12.375	0.030
395	9	5	12.34375	0.08125
12441	14	10	12.3457031	0.000273
12944671	24	20	12.3438072	2.67E-07
414229983	29	25	12.345	1.39E-09
1656919932	31	27	12.345	1.39E-09

Operations:
- Addition (Subtraction):
  - Left-shift the larger number (both numbers have the SAME shift).
  - Add the two numbers.
  - Keep the shift.
- Multiplication (Division):
  - Multiply the two numbers.
  - Add two shifts.
  - Right shift the result to make the result in range.

Blinking LED:

```
int main() {
    initLeds();
#ifdef TimerInt
  configureTimer(period);
  timerFired = false;
  while (1) {
    if (timerFired) {
      timerFired = false;
      ledToggle(1);
    }
  }
#else
  while (1) {
    for (int i=0; i<50; i)
      __delay_ms(20);
    ledToggle(1);
  }
#endif
  return 0;
}
```

Left Shift	<<, <<=	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	Before: 0011 1100 << 2 Result: 1111 0000
Right Shift	>>, >>=	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	Before: 0011 1100 >> 2 Result: 0000 1111

Example | Detecting & Resolving Updating Errors

"load new code to scratch memory"

The best way to detect an error loading new code into scratch memory is to validate the code's checksum before and after code is loaded into scratch memory. If the checksum doesn't match what was recorded prior loading code then an error occured. Attempt to load the code again. The updater can wait as long as necessary to reload the code until the checksum confirms that the code in scratch is valid. If a valid checksum is not reached after 3 attempts then reload last working code into scratch.

"erase code sector"

The best way to detect an error erasing code sector is to check a sector and see if it's been set to 0XFFFFFFF. If you don't see 0xFFFFFFFF then some section of the code sector hasn't been erased. Attempt to erase again. Test contents of a code sector again. If the code sector cannot be erased after 3 attempts then there is a section of the code sector damaged. Update cannot be processed and the issue will need to be resolved though hardware repair.

"program code"

Similar to (4.a) the best way to determine if program code has an error would be to check the checksum to determine if it's valid. If it's not then repeat the "erase code sector" step and try again until a valid checksum is returned. If no valid checksum is returned then reload last working code.

Reentrant Functions

- Race condition could be triggered by calling non-reentrant functions.
- A reentrant function is a function where there is a provision to
  - interrupt the function in the course of execution,
  - service the interrupt service routine, and then
  - resume the earlier going on function,
  - without hampering its earlier course of action.
- Non-reentrant functions
  - Have static or global variables
  - Some C functions: malloc, printf (global variables)

Final Exam Cheat Sheet 1

State	Next State	Event	Action
State 0	State 1	Button Pressed	Start Delay
State 1	State 2	Delay Expired	Led On
State 2	State 3	Button Released	Led On / Start Delay
State 3	State 0	Delay Expired	Led Off

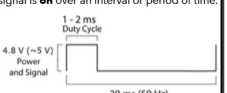
- Delay
  - <util/delay.h>: https://www.nongnu.org/avr-libc/user-manual/group__util__delay.html
  - Define cpu frequency
    - Example: #define F_CPU 4000000UL // 4MHz
  - _delay_ms()
    - Atmega128's max delay is 262.14ms/f_cpu in MHz
    - Example: f_cpu=4MHz, max delay is 262.14/4=65.535ms
  - _delay_us()
    - Max Delay is 768us/f_cpu in MHz.

```
while (1) {
    if (!wasPressed && buttonPressed) {
        ledOn(1);
        __delay_ms(10);
        wasPressed = true;
    } else if (wasPressed && !buttonPressed) {
        ledOff(1);
        __delay_ms(10);
        wasPressed = false;
    }
}
```

Pulse Width Modulation (PWM)

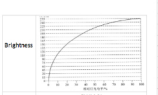
PWM Definition: PWM allows us to vary how much time the signal is high in an analog fashion. While the signal can only be high (usually 5V) or low (ground) at any time, we can change the proportion of time the signal is high compared to when it is low over a consistent time interval.

...This high signal is called the **Duty Cycle (DC)**. Duty cycle is measured in percentage. The percentage duty cycle specifically describes the percentage of time a digital signal is **on** over an interval or period of time.



PWM Applications:
- Dimming LED's.
  - Shorter Duty Cycle = Dimmer.
  - Longer Duty Cycle = Brighter.
- Motor Control:
- RPM's (Slowing):
  - Normal Motor has > 1000RPM's.
  - Motor ONLY rotates during Duty Cycle.
- Servo Motor:
  - 180 Degree (-90 to +90)
    - Directional
    - Bidirectionl
  - Direction -90 + (Duty Cycle -1)*180

Interpret Graph:



PWM%	Brightness
100	10
71	9
58	8
42	7
30	6
21	5
14	4
9	3
4	2
1	1
0	0

Example:
Question: The period of PWM is 20ms. What should be the length of duty in each PWM period to reduce the rotation speed to 10RPM? Assume no mechanical delay in motor on the change of PWM signal.
Answer:

∴ Data Given:
Original RPM: 1000 RPM
Desired RPM: 10 RPM
PWM Period (T): 20ms

∴ From the data given we know that the motor originally completed 1000 RPM for a 20ms duty cycle. Now we want to reduce that to 10RPM for a 20ms period, that's a reduction of 99%, or put another way the motor is now rotating 1% of the time meaning that the rest of the time the motor is dormant.

∴ From lecture we know have the following equation:

System Start Sequence

Boot Loader is a specialized driver that starts and initializes the whole system.

Initialization includes the following items (ATMega128 Specific):
- Setup the *Interrupt Vector Table*.
- Clear SREG (Status Register).
- Set *Stack Pointer*.
- Enable sleep.
- Initialize the .data and .bss sections of memory.
- Configure on-board devices (SPI).
- Enable Interrupts.

Start .app (int main) after *Initializations* have completed the main application is started including:
- Setup.
- Get into the main *While Loop*.

WARNING! Problems with *While Loops*
- Multi-tasking + Interrupts:
  - Multiple components work simultaneously.
  - Tasks share the same resources.
  - Tasks communicate amoung each other.
  - Interupppts change the flow of code
- Race Conditions (see *race conditions*)

Reentrant / Non-reentrant Functons

Reentrant Function Definition: Is a function where there is provision to:
- Interrupt the function in the course of execution,
- service the interrupt service routine and then,
- Resume the earlier going function,
- Without hampering its earlier course of action.

Non-reentrant Function Definition:
- Have static or global variables.
- Common C Library functions that are non-reentrant:
  - Malloc
  - printf (Due to global variables)

Non-reentrant Function Definition:
- Have static or global variables.
- Common C Library functions that are non-reentrant:
  - Malloc
  - printf (Due to global variables)

Function vs. Macro

Function vs. Macro: A macro inserts a line of code into any references to the macro (does not need to jump Anywhere). A function will cause the sp to jump to the function declaration. Generally Functions are better performance wise

```
if a < b < c, return b
else, return c
else,
  if b < c, return b
  else, return c
```

- Function scope
  - Stack is allocated one time
- Block scope
  - Stack is adjusted for each block

```
#define min3(x, y, z) (((x)<(y))?(((x)<(z))?(x):(z)):(((y)<(z))?(y):(z)))
```

Scheduler

Reason for Scheduler:
- Helps resolve *Race Conditions*.
- Provides a way to handle multiple events executing at the same time.
- Handles events 1-by-1.
- Solves *Finite State Machine* issues (see *FSM's*).

Implementation:
- **Overview:**
  - Use a queue for tasks, *Task Queue*.
  - Interrupt *adds* new task to the queue.
  - App *removes* tasks from the queue.
  - Queue opertations are a *Atomic* (see *Race Conditions*).
  - Both *Interrupt* and App have bounded queue operation times.
- **Queues:**
  - Queue Operations (Are Atomic!):
    - initQueue(): Initalize queue
    - isEmpty(): Check if queue is empty.
    - append(): Adds event to queue.
    - pop(): Get next event in the queue.
  - (List-based) Queue
    - Data Structure: List (single-linked or double-linked).
    - Priority: FIFO
    - Problem: Heap Allocation, Unlimited Length, Non-deterministic operation time.
  - (Set-based) Queue
    - Data Structure:
      - Array-based set.
      - Fixed size with allocation in the .data section of memory.
    - Problem: Any event cannot be repeated before being processed first.
- **Coding Tips:**
  - Nested loops in *int main*.
    - Outer Loop: while(1){}
    - Inner Loop: Scheduler while(someCondition){}
  - Scheduler is implmented according to application's requirements.
  - Pop & Amend MUST:
    - run atomically.
    - run with deterministic time.
  - Inner loop executes taks when there are tasks in queue.

Example | Scheduler+Sleep

```
int main() {
    setSleep();
    initQueue(&queue);
    while (1) {
        // queue loop
        while (!isEmpty(&queue)) {
            // process the events in queue
            uint8_t event = pop(&queue);
            handleEvent(event);
        }
        sleep(); }
}
```

Consts vs. Strings

- ALL consts and strings are stored in *Code Space*.
- Do NOT duplicate strings in code. Instead declare a **char** arrays for strings and use pointer to reference the strings.
- Remove auxiliary code:
  - Assert, log, debug.
  - Use Macro to enable/disable auxillary code.

```
#define MOTOR_LOG 1 // set this to zero to turn off debugging
#if MOTOR_LOG
#define Log(level, str)              Log(eMotorSubSystem, (level)
#define LogWithNum(level, str, num) LogWithNum(eMotorSubSystem,
#else
#define Log(level, str)
#define LogWithNum(level, str)
#endif
```

Unoptimized

```
int main() {
  char str[]="hello world!\n";
  printf("hello world!\n");
}
```

Optimized

```
int main() {
  char str[]="hello world!\n";
  printf(str);
}
```

Optimization Levels

What Optimizations Does:
During the process of optimization the compiler will exlude/ reduce standard libraries that are NOT needed including:

.text.ibgcc.mul	*.text.ibgcc.builtins*
.text.ibgcc.div	*.text.ibgcc.fmul*
.text.ibgcc.prologue	*.text.ibgcc.fixed*

Optimization 0 (O0):	*NO optimizations*
Optimization 1 (O1):	*Some Speed Optimization.*
Optimization 2 (O2):	*More Speed Optimization.*
Optimization 3 (O3):	*Most Speed Optimization.*
Optimization S (OS):	*Space Optimization.*

Implementation	1 call (diff from baseline)	2 calls	3 calls
Macro	0	76	152
Function (local or external)	20	60	96
Macro with space optimization	–40	8	56
Function with space optimization	–40	–20	0

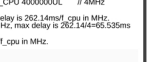
Applications

Basic Applications Needs:
- Clock (Timer).
- Input (Digital / Analog).
- Output (Digital / Analog).

Type of Applications:
- Blinking / Flashing (Output).
- Pulse Width Modulation (PWM).
- LED Dimmer (Bright/ Dim).
- Motor Control (See PWM).
- Debug: Logic Analyzer

Servo motor

- https://www.youtube.com/watch?v=KDfhXd2kell
- 180 degree, bidirectional, direction control
- Direction = -90+(duty-1)*180
- If accuracy is 1 degree,
- Then duty step size is 1/180ms or 5.556us



Template

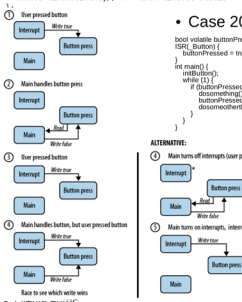
```
case (state):
  if event valid for this state
    handle event
    prepare for new state
    set new state
```

Example

```
switch (state) {
case (green light):
  if (event is stop command)
    turn off green light
    turn on yellow light
    set state to yellow light
    start timer
  break;
```

State-centric


```
struct sStateTableEntry {
  tLight light;       // all states have associated lights
  tState goEvent;     // state to enter when go event occurs
  tState stopEvent;   // ... when stop event occurs
  tState timeoutEvent;// ... when timeout occurs
};
```



• Case 20

```
bool volatile buttonPressed = false;
ISR(_Button) {
  buttonPressed = true;
}
int main() {
  initButton();
  while (1) {
    if (buttonPressed) {
      dosomething(); // race
      buttonPressed = false;
      dosomeotherthing();
    }
  }
}
```

- Event-centric
- Each event transition includes:
  - Next state.
  - Activity.
- Each state-table row includes:
  - Current state.
  - All possible event transitions.
- State-table MUST be initialized before the while() loop
- All events MUST be handled in the while () loop.

Example | Solving Interrupt Problems

A system has a 40MHz processor. It needs to process audio signals with a 40000Hz interrupt.

Question: *How many cycles are between two interrupts?*
Answer: *(Below)*

$$\therefore Period = (Processor)MHz * \frac{1}{(Interrupts)Hz}$$

$$\therefore Period = 40MHz * \frac{1}{40{,}000Hz}$$

$$\therefore Period = 40MHz * (\frac{1{,}000{,}000Hz}{1MHz}) * \frac{1}{40{,}000Hz}$$

$$\therefore Period = 40{,}000{,}000Hz * \frac{1}{40{,}000Hz}$$

$$\therefore Period = 1000 cycles$$

$$\therefore 1000 cycles \textbf{ are needed between two interrupts.}$$

(1/period) * Clock Frequency = Cycles

Question: *Assume 20 cycles are needed for switching context. What is the percentage of time for switching context?*
Answer: *(Below)*

$$\therefore Period = 1000 cycles$$

$$\therefore Switching = 20 cycles$$

$$\therefore \%\ Switching = (\frac{CyclesSwitching}{CyclesPeriod}) * 100$$

$$\therefore \%\ Switching = (\frac{20 cycles}{1000 cycles}) * 100$$

$$\therefore \%\ Switching = 0.02 * 100$$

$$\therefore \%\ Switching = 2.0\%$$

Question: *Assume 380 cycles are needed for the ISR. What is the percentage of time in total for processing an interrupt?*
Answer: *(Below)*

$$\therefore Period = 1000 cycles$$

$$\therefore ISR = 380 cycles$$

$$\therefore \%\ ProcessingInterrupts = (\frac{CyclesISR + CyclesSwitching}{CyclesPeriod}) * 100$$

$$\therefore \%\ ProcessingInterrupts = (\frac{380 cycles + 20 cycles}{1000 cycles}) * 100$$

$$\therefore \%\ ProcessingInterrupts = (\frac{400 cycles}{1000 cycles}) * 100$$

$$\therefore \%\ ProcessingInterrupts = 0.40 * 100$$

$$\therefore \%\ ProcessingInterrupts = 40\%$$

Table 17. Sleep Mode Select — ATMega128

SM2	SM1	SM0	Sleep Mode
0	0	0	Idle
0	0	1	ADC Noise Reduction
0	1	0	Power-down
0	1	1	Power-save
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Standby[1]
1	1	1	Extended Standby[1]

Example | Solving FSM Queue

Prompt: *A Queue is designed to be deterministic with a fixed maximum length. The struct of Queue is below: a currEvent and an array. {currEvent, [[0: Ex], [1: Ex], [2: Ex], [3: Ex]]}*

An example of Queue is below.
{E3, [[0: END], [1: E2], [2: END], [3: E1]]}

It means the current pending event is E3. The following events in order are E1 and E2. The currEvent E3 points to [3: E1], where 3 is the index of the array and E1 means the next event E1.

Then, the next event E1 points to [1: E2], where 1 is the index of the array and E2 means the next event E2.

Then, the next event E2 points to [2: END], where 2 is the index of the array and END means the end, i.e. no more events.

When the Queue has no events, it has the following:
{END, [[0: END], [1: END], [2: END], [3: END]]}

Question: *Assume in the system, the event following time order:*

T0: no events in queue.
T1: E1 is added to the queue.
T2: E0 is added to the queue.
T3: E3 is added to the queue.
T4: E1 is popped from the queue.
T5: E2 is added to the queue.

Answer:

Time	currEvent	[0: x]	[1: x]	[2: x]	[3: x]
T0	END	[0: END]	[1: END]	[2: END]	[3: END]
T1	E1	[0: END]	[1: END]	[2: END]	[3: END]
T2	E1	[0: END]	[1: E0]	[2: END]	[3: END]
T3	E1	[0: E3]	[1: E0]	[2: END]	[3: END]
T4	E0	[0: E3]	[1: END]	[2: END]	[3: END]
T5	E0	[0: E3]	[1: END]	[2: END]	[3: E2]

Interrupts

Interrupt Procedure:
1. An *Interrupt Request* (IRQ) happens inside the processor, triggered by a peripheral, software, or an error.
2. The processor saves the context (status of execution prior to interrupt).
3. The processor looks in the interrupt vector table to fint the *callback function* (ISR) associated with the interrupt.
4. The callback function runs.
5. The processor resotre the context and resumes (execution) from where it was interrupted.

```
HandyHelperFunction:
  disable interrupts
  do critical things
  enable interrupts

CriticalFunction:
  disable interrupts
  call HandyHelperFunction
  do supposedly critical things
  enable interrupts
```

```
HandyHelperFunction:
  interrupt status = disable interrupts()
  do critical things
  enable interrupts(interrupt status)

CriticalFunction:
  interrupt status = disable interrupts()
  call HandyHelperFunction
  do critical things
  enable interrupts(interrupt status)
```

CSXP

Interrupt Priority:
- Determine which ISR to be called when multiple interrupts occur.
- Determine whether an interrupt can supersede the one currently running.

Non-maskable:
- Interrupts that CANNOT be disabled.
- Non-maskable Interrupts (NMI's):
  - Processor exceptions.
  - Reset.

Interrupt Requests (IRQ's):
- Interrupts MUST be configured and enabled. Can be configured or enabled by the compiler's startup code or software, and should ONLY be enabled when needed.
- Setup:
  - Disable interrupts first.
  - Next configure interrupt control registers.
- Reduce Interrupt Overhead:
  - Only save the registers that are written by ISR.
  - Only use inline functions.
  - Only process data that must be processed.

Callback Functions (ISR's):
- Initialize the *Interrupt Vector table* (IVT). The IVT is setup by the bootloader (see *System Startup Process*).
- Call IST:
  - Make IST short to reduce latency.
  - Do not call non-reentrant functions.
  - Turn off ALL other interrupts.
  - Enable vs. Disable ISR's.

Disable
- Save interrupt bit
- Set interrupt bit to 0

```
mk=(1<<SREG_I);
st=SREG&mk;
SREG&=~mk;
```

Enable
- Restore interrupt bit

```
SREG|=st;
```

Pros vs. Cons of Interrupts:
- Pros:
  - A system is time-critical.
  - An event is expensive to check for and happens very rarely.
  - A short background task lets the system run more smoothly.
- Cons:
  - Incur overhead.
  - Make the system less deterministic.
  - Hard to track down bugs.
  - Need processor-dependent configuration.

Non-interrupts:
- Polling:
  - Last option to consider.
  - Need a time-out in case polling does not get respond in time.
- System Tick
  - For periodic applications.
  - Must process everything during the interval of two ticks.

Interrupt Based Code Flow

4 Things To Optimize

1. ***Code Space:***
   - *Linker Script:*
     - *Overview:*
     - *Layout:*
       - *BSS Segment (.bss)*
         - *Contains uninitialized global variables that will go RAM.*
       - *DATA Segement (.data)*
         - *Contains global variables that are initialized and will go in RAM.*
         - *May includ .bss as a subsegment.*
         - *May include the Heap and Stack.*
       - *TEXT Segment (.text)*
         - *Contains code and constant data that may be put in read-only memory or in RAM.*
     - *Vector Table*
       - *Containts the exception vector table to handle interrupts.*
   - *Map File*

```
while (1) {
  pollButton();

switch (st) {
    case S0_Released:
      //ledOff(1);
      if (evButton == Button_Pressed) {
        ledOn(1);
        startDelay();
        st = S1_Pressed;
      }
      break;
    case S1_Pressed:
      //ledOn(1);
      if (evTimer == Delay_Expired) {
        st = S2_Pressed;
      }
      break;
    case S2_Pressed:
      // ledOn(1);
      if (evButton == Button_Released) {
        ledOff(1);
        startDelay();
        st = S3_Released;
      }
      break;
    case S3_Released:
      // ledOff(1);
      if (evTimer == Delay_Expired) {
        st = S0_Released;
      }
      break;
    default:
      // only for errors to reset to init
      st = S0_Released;
      evButton = NONE;
      evTimer = NONE;
  }
```

2. ***Data RAM:***
   - *Malloc:*
     - *Do NOT use!*
     - *Wasted RAM, lost processor cycles, and fragmentations.*
     - *Non-deterministic operations.*
     - *Non-deterministic RAM size (depending or state of the Heap).*
     - *Alternatives to Malloc:*
       - *Declare global variables.*
       - *Recycle buffers.*
       - *Circular buffer.*
       - *Chunk allocations.*
   - *From (Map) determine:*
     - *DATA Segment (.data).*
     - *BSS Segment (.bss).*
   - *Stack:*
     - *The size of the Stack is determined by:*
       - *Local variables.*
       - *Chain on function calls.*
     - *Stack* is allocated one ONE time.
     - *Do NOT use recursion.*
     - ***Macros do NOT use Stack space and are fa than functions.***
3. ***Speed:***
   - *Profiling (Definition): To know where the cycles are goin.*
   - *Profiler (Definition): Changes the behavior and timing of the code.*
   - *I/O Lines.*
   - *Ways of Measuring Speed:*
     - ***Timer:*** *In case there's no stopwatch to use get time before and after function call (to function we want to measure) subtract the end time from the start time to determine execution time of desired function.*
     - ***Sampling:***
       - *Procedure:*
         - *Set a periodic timer and a block of RAM.*
         - *One each timer, save the return address of the Stack.*
         - *When the RAM is full, output the list of addresses.*
         - *Figure out where the addresses are in the image using the map file.*
       - *Requirements:*
         - *The sampling timer is the only one allowed to interrupt other interrupts*
4. ***Power:***

Power Consumption:
- *Power:*
  - *Equation:* ***Power = Voltage * Current***
  - *Units:*
    - *Power: Watt (W)*
    - *Current: Ampere (A)*
    - *Voltage: Joules (J)*
- *Energy:*
  - *Equation:* ***Energy = Power * Time***
  - *Equation #2:*

Operational Energy = (OpPower +OpTime) + IdlePower + IdleTime)
- *Units:*
  - *Power: Watt (W)*
  - *Time: Seconds (Sec)*
  - *Energy: Jourles (J)*

$$P = J*A$$

$$E = P*T$$

Problems *(with Power Consumption):*
- *Heat:*
  - *Determined by Power (of system).*
  - *Can cause inconsistencies in ALU calculations if system is running too hot.*
- *Battery:*
  - *Determined by Energy (coming/going).*
  - *Processor requires specific size battery to fulfill operating timeframe between changing batteries.*

Solutions *(to Power Consumption):* ***[Shown Below]***

Turn off Components	*When components are NOT needed:*
	- *Turn OFF peripherals:*
	  - *Power down unnecessary LED's.*
	  - *Power down unused sensors.*
	- *Turn OFF unused I/O pins:*
	  - *Set I/O pins to input with internal pulldown.*
	  - *Set I/O pins to tristate.*
	- *Turn OFF process subsystems.*
	  - *Power down Serial Peripheral Interface (SPI).*
	  - *Power down Universal Synchronous and Asynchronous Receiver-Transmitter (USART).*	

Slow Down	Clock Switching:
	- Fast and slow clock sources depending on task's needs.
	Frequence Scaling:
	- Slower clock means smaller power (NOT less energy).
	- *Dynamic Voltage Scaling* (DVS) is used to scale processor frequency.
	- Assume all taks are known with worst execution time Ci and period Pi under maximum frequency F_{max}.
	  - Tasks are schedulable @ F_{max} without DVS if $Sum \frac{Ci}{Pi} < 1$.
	  - Tasks are schedulable @ F_{scaled} with DVS if $Sum \frac{Ci}{Pi} < \frac{Fs}{Fm}$.
	- $F_{min} = F_{max} * Sum \frac{Ci}{Pi}$.
		$Sum \frac{Ci}{Pi} < 1$
		$Sum \frac{Ci}{Pi} < \frac{Fs}{Fm}$

Sleep	Common *Sleep* Configurations:
	- Slow Down:
	  - Slow the clock down to hundred of Hertz.
	- Idle or Sleep:
	  - Turn OFF the processor's cores but keep enabled timers, peripherals, and RAM active.
	- Deep Sleep (or Light Hibernation):
	  - In addition to the processor's cores being turned OFF, soem (possible ALL) on-chip peripherals can be configured to turn OFF.
	  - Subsystems that generate interrupts are on to wake the processor's cores.
	- Deep Hibernation (or Power Down):
	  - The processor chooses which on-chip perpherals to be turned OFF (usuallal ALL). RAM is often left in an unstable state.
	  - Processor registers are retained, so the system deosn't need to intialize on boot.
	  - A wakeup pin or a small subste of interrupts can restart the processor.
	- Power Down:
	  - The processor does not retain an memory, and starts from a completely clean state.
	Issues with *Sleep:*
	- Going into *Sleep* causes *Wakeup* latency.
	- Must maintain *Wakeup* mechanisms to wake system from *Sleep*.
	- Deeper sleep modes place the processor intoa lower-power state but take longer to wake from which increases system latency.
	- The *Wakeup* interrupts could be buttons, timers, other chips (E.g.g ADC conversion complete), or traffic on a communication bus.	

Power Saving Examples	ATMega128	PIC24EP512GU810
	- Idle - ADC Noise Reduction - Power-down - Power-save - Standby & Extended Standby	- Clock Switching - Idle - Sleep - Doze - Peripheral Disable