Sensor Processing

So far, our code looks something like this:

```
while(1) {
    <read some sensors>
    <respond to the sensor input>
    <read some other sensors>
    <respond to the sensor input>


}
```

Sensor Processing

- Sometimes, this is sufficient
- Other times:
  - We need to respond to certain events very quickly, or
  - We need to time events very carefully

Interrupts

- Hardware mechanism that allows some event to temporarily interrupt an ongoing task
- The processor then executes a small piece of code called: **interrupt handler** or **interrupt service routine** (ISR)
- Execution then continues with the original program

Some Sources of Interrupts (atmega2560)

External:

- An input pin changes state
- The UART receives a byte on a serial input

Internal:

- A clock
- Processor reset
- The on-board analog-to-digital converter completes its conversion

Interrupt Example

Suppose we are executing code from your main program:

```
LDS R1 (A) ← PC
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3
```

An Example

Suppose we are executing code from your main program:

```
LDS R1 (A)
LDS R2 (B) ← PC
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3
```

An Example

Suppose we are executing code from your main program:

```
LDS R1 (A)
LDS R2 (B)
CP R2, R1  ← PC
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3
```

An Example

An interrupt occurs (EXT_INT1):

```
LDS R1 (A)
LDS R2 (B)
CP R2, R1   ← PC
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3
```

An Example

Execute the interrupt handler

```
  LDS R1 (A)
  LDS R2 (B)
  CP R2, R1
▶ BRGE 3      ← remember this location
  LDS R3 (D)
  ADD R3, R1
  STS (D), R3
```

An Example

Execute the interrupt handler

```
  LDS R1 (A)
  LDS R2 (B)
  CP R2, R1
▶ BRGE 3
  LDS R3 (D)
  ADD R3, R1
  STS (D), R3
```

```
EXT_INT1:

PC → LDS R1 (G)
     LDS R5 (L)
     ADD R1, R2
          :
     RETI
```

An Example

Execute the interrupt handler

```
LDS R1 (A)
LDS R2 (B)
CP R2, R1
▶ BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3
```

```
EXT_INT1:

      LDS R1 (G)
PC →  LDS R5 (L)
      ADD R1, R2
           :
      RETI
```

An Example

Execute the interrupt handler

```
  LDS R1 (A)
  LDS R2 (B)
  CP R2, R1
▶ BRGE 3
  LDS R3 (D)
  ADD R3, R1
  STS (D), R3
```

```
     EXT_INT1:

     LDS R1 (G)
     LDS R5 (L)
PC → ADD R1, R2
         :
     RETI
```

An Example

Execute the interrupt handler

```
  LDS R1 (A)
  LDS R2 (B)
  CP R2, R1
▶ BRGE 3
  LDS R3 (D)
  ADD R3, R1
  STS (D), R3
```

```
        EXT_INT1:

        LDS R1 (G)
        LDS R5 (L)
        ADD R1, R2
PC →        :
        RETI
```

An Example

Return from interrupt

```
  LDS R1 (A)
  LDS R2 (B)
  CP R2, R1
▶ BRGE 3
  LDS R3 (D)
  ADD R3, R1
  STS (D), R3
```

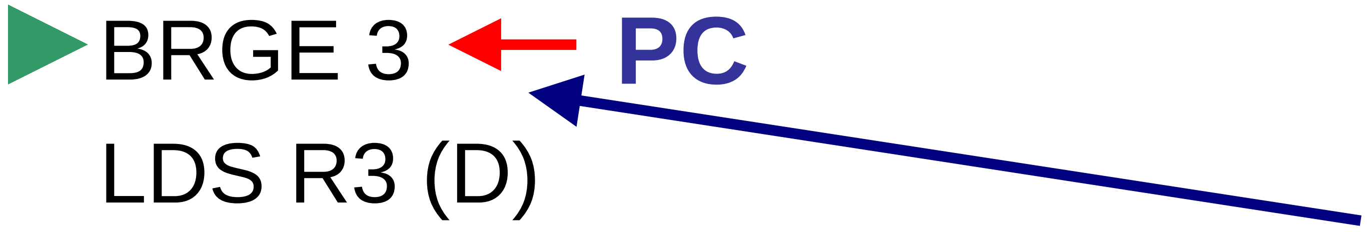
```
      EXT_INT1:

      LDS R1 (G)
      LDS R5 (L)
      ADD R1, R2
          :
PC →  RETI
```

An Example

Return from interrupt

```
LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3
```



```
EXT_INT1:

LDS R1 (G)
LDS R5 (L)
ADD R1, R2
      :
RETI
```

An Example

Continue execution with original

```
LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)  ← PC
ADD R3, R1
STS (D), R3
```

```
EXT_INT1:

LDS R1 (G)
LDS R5 (L)
ADD R1, R2
      :
RETI
```

An Example

Continue execution with original

```
LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1 ← PC
STS (D), R3
```

```
EXT_INT1:

LDS R1 (G)
LDS R5 (L)
ADD R1, R2
      :
RETI
```

Interrupt Service Routines

Generally a very small number of instructions

- We want a quick response so the processor can return to what it was originally doing
- No delays, waits, or floating point operations(**) in the ISR…

Timer-Based Interrupts

- Interrupt source: internal hardware timer
- This allows us to produce an interrupt at some regular period

- The exact mechanism is different depending on the type of processor you are using (even if you are using the Arduino environment)

Teensy: Timer1

“Timer1” is one predefined variable that can be configured to handle timer operations. Key ones include:

- `Timer1.initialize(usec)`: initialize the timer and set its period
- `Timer1.attachInterrupt(func)`: configure the timer to execute **func** once every period
- `Timer1.start()`: start running the timer

Timer Example

What does this program do?

```
#include <TimerOne.h>

void myISR()
{
  GPIOC_PDOR ^= 0x20;
}

void setup() {
  // Configure PORTC, bit 5 to be a digital I/O bit
  PORTC_PCR5 = PORT_PCR_MUX(0x1);
  // Configure bit 5 to be an output
  GPIOC_PDDR = 0x20;

  // Configure the timer
  Timer1.initialize(200000);
  Timer1.attachInterrupt(myISR);
  Timer1.start();
}

void loop() {
}
```

Timer Example

- `myISR()` is called every 200 ms
- Each call to this function flips the state of the built-in LED
- So: the LED flashes at 2.5 Hz

- Note that this happens even though loop() does nothing!
  - The ISR executes asynchronously from loop()

Timer Example II

What does this program do?

```
void myISR()
{
  static int counter = 0;
  ++counter;
  if(counter == 5) {
       GPIOC_PDOR ^= 0x20;
       counter = 0;
  }
}

void setup() {
  PORTC_PCR5 = PORT_PCR_MUX(0x1);
  GPIOC_PDDR = 0x20;

  // Configure the timer
  Timer1.initialize(200000);
  Timer1.attachInterrupt(myISR);
  Timer1.start();
}

void loop() {
}
```

Timer Example II

- LED flips state once every fifth call to the ISR
- So: the flashing frequency is 2.5/5 = 0.5 Hz

Timer1 Notes

Timer1 is used within the Arduino Environment to handle analogWrite() for pins 3 and 4 (for the Teensy 3.5)

- By using the timer, analogWrite() will not longer function
- Instead, you can use: Timer1.pwm(pin, duty) to configure PWM for pins 3 and 4
- And Timer1.setPwmDuty(pin, duty) to change the duty cycle
- Note duty = [0 … 1023]

Timer1: Other Functions

- `Timer1.stop()`: stop the timer
- `Timer1.resume():` continue the timer
- `Timer1.restart():` start the timer at the beginning of the period
- `Timer1.detachInterrupt():` turn off the ISR

Timer3

Timer3 behaves the same way as Timer1

- Arduino pins 29 & 30 on the Teensy 3.5

Controlling LED Brightness

What is the relationship of current flow through an LED and the rate of photon emission?

- They are linearly related (essentially)

Controlling LED Brightness

Suppose we pulse an LED for a given period of time with a digital signal: what is the relationship between pulse width and number of photons emitted?

Controlling LED Brightness

Suppose we pulse an LED for a given period of time with a digital signal: what is the relationship between pulse width and number of photons emitted?

- Again: they are linearly related (essentially)

- If the period is short enough, then the human eye will not be able to detect the flashes

Timer Example III

- Problem: implement an ISR that generates a PWM signal
- The duty cycle is determined by the state of a global variable (“duty”)

Timer Example III

```
volatile uint8_t duty = 0;

void loop() {
  for(int i = 0; i < 255; ++i) {
    duty = i;
    delay(10);
  }
  for(int i = 255; i > 0; --i) {
    duty = i;
    delay(10);
  }
}
```

What is the ISR implementation?

Timer Example III

```
void setup() {
  PORTC_PCR5 = PORT_PCR_MUX(0x1);
  GPIOC_PDDR = 0x20;

  // Configure the timer
  Timer1.initialize(100);
  Timer1.attachInterrupt(myISR);
  Timer1.start();
}
```

Timer Example III

```
void myISR()
{
    static uint8_t counter = 0;
    ++counter;
    if(counter < duty)
        GPIOC_PDOR |= 0x20;
    else
        GPIOC_PDOR &= ~0x20;

}
```

Timer Example III

```
void myISR()
{
  static uint8_t counter = 0;
  ++counter;
  if(counter < duty)
    GPIOC_PDOR |= 0x20;
  else
    GPIOC_PDOR &= ~0x20;
}
```

PWM Implementation

What is the resolution (how long is one increment of “duration”)?

PWM Implementation

What is the resolution (how long is one increment of “duration”)?

- 100 usecs

PWM Implementation

What is the period of the pulse?

PWM Implementation

What is the period of the pulse?

- 100 usecs * 256 = 25.6 ms

NOTE: DON’T USE THIS SOFTWARE PWM FOR YOUR PROJECT

- Use hardware PWM instead

Interrupt Service Routines

- Should be **very** short
  - No “delays”
  - No busy waiting
  - Function calls from the ISR should be short also
  - Minimize looping
  - No “printf()”
- Communication with the main program using ***volatile*** global variables

Interrupts, Shared Data and Compiler Optimizations

- Compilers (including ours) will often optimize code in order to minimize execution time
- These optimizations often pose no problems, but can be problematic in the face of interrupts and shared data

Shared Data and Compiler Optimizations

For example:

```
A = A + 1;
C = B + A
```

Will result in ‘A’ being fetched from memory once (into a general-purpose register) – even though ‘A’ is used twice

Shared Data and Compiler Optimizations

Now consider:

```
while(1) {
   GPIOB_PDOR = A;
}
```

What does the compiler do with this?

Shared Data and Compiler Optimizations

The compiler will assume that ‘A’ never changes.

This will result in assembly code that looks something like this:

```
R1 = A;  // Fetch value of A into register 1
while(1) {
    GPIOB_PDOR = R1;
}
```

The compiler only fetches A from memory once!

Shared Data and Compiler Optimizations

This optimization is generally fine – but consider the following interrupt routine:

```
myISR(){
  A = GPIOC_PDIR;
}
```

Shared Data and Compiler Optimizations

This optimization is generally fine – but consider the following interrupt routine:

```
myISR(){
  A = GPIOC_PDIR;
}
```

- The global variable ‘A’ is being changed!
- The compiler has no way to anticipate this

Shared Data and Compiler Optimizations

The fix: the programmer must tell the compiler that it is not allowed to assume that a memory location is not changing

- This is accomplished when we declare the global variable:

```
volatile uint8_t A;
```

Shared Data and Compiler Optimizations

volatile uint8_t A;

This will cause the compiler to do this:

```
  while(1) {
     R1 = A;  // Fetch value of A into reg 1
     GPIOC_PDOR = R1;
  }
```

The compiler fetches A from memory every time it needs it!

Shared Data and Interrupts

- Recall: the data bus on the Atmel mega2560 is 8 bits wide
- A byte can be transferred in one cycle
- Any data structure larger than a byte requires multiple transfers

When there are interrupts: this can lead to subtle (but very real) problems

For example:

```
uint16_t a;
a = a + 5;
```

For example:

```
uint16_t a;
a = a + 5;
```

Steps:

- Transfer of the low byte from memory to a general purpose register
- Transfer of the high byte
- Addition operation (multiple steps)
- Transfer of the low byte from GP to mem
- Transfer of the high byte from GP to mem

Suppose that an ISR routine views and then modifies the variable a …

- Transfer of the low byte from memory to a general purpose register
- Transfer of the high byte
- Addition operation (multiple steps)
- Transfer of the low byte from GP to mem
- Transfer of the high byte from GP to mem

- Transfer of the low byte from memory to a general purpose register
- Transfer of the high byte
- Addition operation (multiple steps)
- Transfer of the low byte from GP to mem
- Transfer of the high byte from GP to mem

Interrupt occurs:

- ISR changes a, but main program still uses old value

- Transfer of the low byte from memory to a general purpose register
- Transfer of the high byte
- Addition operation (multiple steps)
- Transfer of the low byte from GP to mem
- Transfer of the high byte from GP to mem

- Transfer of the low byte from memory to a general purpose register
- Transfer of the high byte
- Addition operation (multiple steps)
- Transfer of the low byte from GP to mem
- Transfer of the high byte from GP to mem

Interrupt occurs:

- The ISR “sees” the new value of the low byte and the old value of the high byte

Solution?

One possibility:

- If the main program is working with a, then it can temporarily disable interrupts while it does this operation
- Note: it should not disable interrupts for very long

Turning off Interrupts

```
volatile uint16_t a;
:
:

noInterrupts();   // Turn off interrupts
a = a + 5;
interrupts();     // Turn them back on
```

Shared Data Problems

- Any time that the main program and the ISR both view/change a global variable, the potential exists for these *shared data problems*
- Always a problem if the variable is larger than the width of the data bus (called a “word”)
- Some single word variables are a problem, but not all are (it depends on how they are used)

Turning off Interrupts

- Always turn off for the shortest time possible
- There are some cases in which interrupts do not need to be turned off for things to work properly

Book Example

```
volatile unsigned char TimerFlag=0;

void TimerISR() {
   TimerFlag = 1;
}

void main() {
   B = 0; // Init outputs
   TimerSet(1000);
   TimerOn();
   BL_State = BL_SMStart;
   TL_State = TL_SMStart;
   while (1) {
      TickFct_BlinkLed();     // Tick the BlinkLed synchSM
      TickFct_ThreeLeds();    // Tick the ThreeLeds synchSM
      while (!TimerFlag){}    // Wait for timer period
      TimerFlag = 0;          // Lower flag raised by timer
   }
}
```

What is happening with the ISR?

Book Example

```
volatile unsigned char TimerFlag=0;

void TimerISR() {
   TimerFlag = 1;
}


void main() {
   B = 0; // Init outputs
   TimerSet(1000);
   TimerOn();
   BL_State = BL_SMStart;
   TL_State = TL_SMStart;
   while (1) {
      TickFct_BlinkLed();     // Tick the BlinkLed synchSM
      TickFct_ThreeLeds();    // Tick the ThreeLeds synchSM
      while (!TimerFlag){}    // Wait for timer period
      TimerFlag = 0;          // Lower flag raised by timer
   }
}
```

- TimerFlag is set to 1 every 1ms
- Acts as a gate for the while loop
- The loop executes once per 1ms

System Safety

Getting embedded code right is hard

- Complex interaction of many pieces
- We often have to test in the real-time context
  - Limited ability to “see” the state of our program
  - A bug can only occur in a very specific situation

System Safety

In practice, it is very difficult to write a program that behaves appropriately in all situations

- In some cases: the program produces incorrect behavior (completely or in part), but continues to execute
- In other cases: the program might “lock-up” and cease to execute critical pieces of code

System Degradation over Time

With use, an embedded system can degrade due to mechanical or electrical variation (or interaction with high-energy particles)

- Electrical connections between components can be broken
- Components can fail
- Memory can be corrupted

Corruption of Memory

Software rot: small changes are made to the program at the machine code level

- Introduces subtle bugs that can lead to incorrect behavior or processor lock-up

Permanent data storage corruption:

- EEPROM might store parameters that affect behavior (e.g., Kp & Kv)
- Corruption also leads to incorrect behavior

Reducing Problems

Proper mechanical stability

- Appropriate choice of connection between components (this includes soldering)
- Strain relief of wires
- Housings for electronics (in some cases, these will reduce the sensitivity to vibrations)

Reducing Problems

Proper electrical stability

- Some component require power supplies to be very clean (very little variation in supplied voltage)
- Some components (e.g. motors) can cause a lot of noise on the power supply
- Electrical isolation is often necessary

Mitigation in the Long Term

Program and data corruption:

- Processors need some way to restore their state to a “factory configuration”
- Most often: a human maintainer will need to “reflash” the memories stored in EEPROM
- But: some systems can autonomously detect when corruption occurs and take steps to correct the corrupted memory

Mitigation in the Short Term

Mission critical systems: build in redundancies

- Multiple copies of a sensor or actuator
- Multiple processors, all performing the same functions (in some cases, the processors are executing different implementations of the same code)
  - Subsystems are responsible for comparing the results across the different copies and choosing which to believe
  - Errors can be detected very quickly, and the embedded system can take appropriate corrective measures

Mitigation in the Very Short Term

Detecting system lock-ups: watchdog timers

- In most embedded systems, we expect certain tasks to be executed at certain rates
- A bug in the code can result in a full stop of the program or in an infinite loop for a condition that is never met

Watch-Dog Timers

Hardware component:

- A short term counter attached to the system clock
- Compare the counter against some fixed threshold, raising an interrupt when they are equal

Watch-Dog Timers

Software component:

- Main program: “feed the dog” periodically by the resetting the counter
- Interrupt service routine: cause a full or partial system reset
  - ISR can use knowledge of the system to attempt a recovery or identify where an error occurs

Watchdogs in Practice

Initialization:

- Register ISR

```
    extern void isr_function();
         :
    wdt_isr(isr_function);
```

- Declare watchdog timeout period

```
    wdt_enable(WDT0_2S);
```

Note: Exact implementation will depend on the processor

Watchdogs in Practice

Use:

- Always execute:

```
    wdt_reset();
```

  within the watchdog period
- ISR function:
  - Clean up after the error
  - Store data for later reporting of the error

Unstable Power Supplies

An unstable power supply can throw a processor into a strange, inconsistent state

- At this point, the results from executing individual instructions can be very uncertain
- Would like the processor to protect itself in these situations

Unstable Power Supplies

A common solution: Brown-Out Detection circuitry

- At minimum, will force a clean reset of the processor before the power supply voltage drops below a critical level
- In some architectures, the processor can be configured to raise an interrupt following a brown-out