

# Sensor Processing

So far, our code looks something like this:

```
loop()
{
    <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

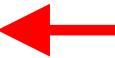
STS (D), R3

# 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

LDS R3 (D)

ADD R3, R1

STS (D), R3

remember this location

# An Example

Execute the interrupt handler

EXT\_INT1:

LDS R1 (A)

LDS R2 (B)

CP R2, R1

► BRGE 3

LDS R3 (D)

ADD R3, R1

STS (D), R3

PC

LDS R1 (G)

LDS R5 (L)

ADD R1, R2

:

RETI

# An Example

Execute the interrupt handler

EXT\_INT1:

LDS R1 (A)

LDS R2 (B)

CP R2, R1

► BRGE 3

LDS R3 (D)

ADD R3, R1

STS (D), R3

PC → LDS R5 (L)

ADD R1, R2

:

RETI

# An Example

Execute the interrupt handler

EXT\_INT1:

LDS R1 (A)

LDS R2 (B)

CP R2, R1

► BRGE 3

LDS R3 (D)

ADD R3, R1

STS (D), R3

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

EXT\_INT1:

LDS R1 (A)

LDS R2 (B)

CP R2, R1

► BRGE 3

LDS R3 (D)

ADD R3, R1

STS (D), R3

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 ← PC

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 or waits 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

```
#include <TimerOne.h>

void myISR()
{
    GPIOC_PDDR ^= 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

What does this program do?

# 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()`

```
void myISR()
{
    static uint8_t counter = 0;
    ++counter;
    if(counter == 5) {
        GPIOC_PDDR ^= 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

What does this program do?

# 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 no 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?

# 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()
{
    // PORTC, bit 5
    static uint8_t counter = 0;
    ++counter;
    if(counter == 0)
        GPIOC_PDOR |= 0x20;
    if(counter >= duty)
        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?

- $100 \text{ usecs} * 256 = 25.6 \text{ ms}$

## NOTE: DON'T USE THIS SOFTWARE PWM FOR YOUR PROJECTS

- Use hardware PWM instead (what you have already been doing)

# Shared Data and Interrupts

- Unlike our Teensies: 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?

# 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

# Another ISR Example...

```
volatile unsigned char TimerFlag=0;  
  
void TimerISR() {  
    TimerFlag = 1;  
}  
  
void main() {
```

# Book Example

What is happening with the ISR?

```
    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  
    }  
}
```

```
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  
    }  
}
```

# Book Example

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

# Example V

Challenge: write an ISR that counts the width of a pulse

- Useful for measuring things like the turning speed of a motor shaft

# Timer Example V: Initialization

```
void setup() {  
    // Configure input pin  
    PORTC_PCR5 = PORT_PCR_MUX(0x1);  
  
    // Configure the timer  
    Timer1.initialize(10);  
    Timer1.attachInterrupt(myISR);  
    Timer1.start();  
}
```

# Timer Example V: Use

```
volatile uint64_t width = 0;

void loop()
{
    noInterrupts()
    uint64_t local_width = width;
    interrupts()
    if(local_width > THRESHOLD + 100)
        <Move motor faster>
    else if(local_width < THRESHOLD)
        <Move motor slower>
}
```

# Example V: ISR

Need to detect changes in PORTC, pin 5

- A variety of solutions are possible
- But: let's use a FSM!

# Timer Example V

```
volatile uint64_t width = 0;

void myISR()
{
    static ISR_STATE state = ISR_LOW;
    static uint64_t counter = 0;
    switch(state)
    {

    }

}
```

# Timer Example V: Epilog

- The Teensy has a 32-bit data bus
  - The transfer of our 64-bit int requires multiple transfers over the data bus
  - So, we must be careful with access
- ISR only writes a correct value to the shared global variable (width)
- Loop only reads from the shared variable when interrupts are disabled
  - But: we must disable for a very short period of time

# Other Questions

- What is the resolution of our “width” variable?
- What is the widest pulse that we can measure?

# Timer Example VI

Suppose that a digital sensor that indicates the failure of a subsystem is connected to a Teensy pin

- We would like to have an emergency procedure initiated as soon as the failure is detected

# One possibility ...

```
loop()
{
    if( !(GPIOA_PDIR & 0x8000) ) {
        initiate_braking();
    }
    < DO OTHER TASKS >
}
```

# ISR Solution

```
#define FAIL_SENSOR_PIN 27

void fail_isr()
{
    // Initiate safe braking
    initiate_braking();
}

void setup() {
    // Generate interrupt on the high-to-low transition
    attachInterrupt(digitalPinToInterrupt(FAIL_SENSOR_PIN),
                    fail_isr, FALLING);
}
```



# Many Challenges to Building Robust Systems

# Coding Challenges

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 that only comes up rarely

# Coding Challenges

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.,  $K_p$  &  $K_v$ )
- 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 components 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
  - We do this on the hovercrafts!

# 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

## System lock-ups

- In most embedded systems, we expect certain tasks to be executed at certain rates
- However, 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 solution:

- 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
- If the “dog is not fed” in a specified duration, then the Interrupt service routine is called
  - ISR can use knowledge of the system to attempt a recovery or identify where an error occurs

# Watchdogs in the Teensies

## 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 can:
  - Clean up after the error
  - Store data for later reporting of the error
  - Reboot the processor

# Dealing with 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

# Mitigating 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