EE 109 Unit 13 – Exceptions and Interrupts
Disclaimer 1

• This is just an introduction to the topic of interrupts. You are not meant to master these right now but just start to use them

• We will cover more about them as we investigate other modules that can make use of them
Exceptions

• Things don’t always go as expected
  – Power goes out
  – Access to wrong area of memory
  – An external event (i.e. a button press or external signal) requires attention from the processor

• Exception Processing
  – Handling tasks that don’t occur as part of the normal sequence of execution of the program ("exceptional events")
  – Unexpected events
  – Expected events that happen at unknown times
  – Completion of tasks we start, but don’t know how long they will take to finish
An Analogy

• Scenario:
  – You're studying (i.e. listening to music and browsing the Internet) but all of a sudden you get a message on your messenger service. What do you do?
  – You stop what you're doing and message back
  – When you're done you go back to studying (i.e. playing a video game or watching YouTube)

• This is what computers do when an exception/interrupt occurs
Interrupts

• Program must respond to events that occur at random times
  – Solution: Let the event “interrupt” the program

• Normal flow of the program is interrupted so it can perform some other action
  – Stop what the program is doing
  – Go deal with the event
  – Then resume executing the program

• Different from calling a function
  – We know when we will call a function (i.e. when we get to that line of code)
  – But interrupts can happen at any time
Recall: Instruction Cycle

- Processor performs the same 3-step process over and over again as it executes a software program
  - **Fetch** an instruction from memory
  - **Decode** the instruction
    - Is it an ADD, SUB, etc.?
  - **Execute** the instruction
    - Perform the specified operation

- This process is known as the **Instruction Cycle**
Recall: Instruction Cycle

- There's actually a 4\textsuperscript{th} step
- After finishing each instruction the processor checks for certain external events or internal error conditions (a.k.a. an \textit{exception})
  - Fetch an instruction from memory
  - Decode the instruction
    - Is it an ADD, SUB, etc.?
  - Execute the instruction
    - Perform the specified operation
  - Check for exceptions
    - If so, pause the current program and go execute other software to deal with the exception
What are Exceptions?

• **Definition**: Any event that causes a break in normal execution
  – "Exceptions" is a broad term to catch many kinds of events that *interrupt* normal software execution

• **Examples**
  – **Hardware Interrupts / External Events [Focus for today]**
    • PC: Handling a keyboard press, mouse moving, USB data transfer, etc.
    • Arduino: Value change on a pin, ADC conversion done, Timers, etc.
  – **Error Conditions [Focus for some other time]**
    • Invalid address, illegal memory access, arithmetic error (e.g. divide by 0)
  – **System Calls / Traps [Focus for some other time]**
    • User applications calling OS code
Interrupt Exceptions

- Two methods for processor and I/O devices to notify each other of events
  - Polling “busy” loop (responsibility on proc.)
    - Processor has responsibility of checking each I/O device
    - Many I/O events happen infrequently (1-10 ms) with respect to the processors ability to execute instructions (1-100 ns) causing the loop to execute many times
  - Interrupts (responsibility on I/O device)
    - I/O device notifies processor only when it needs attention

Recall: Once the A-to-D converter has started we need to wait until the ADSC bit is 0 (i.e. keep waiting 'while' ADSC bit is 1)

With Interrupts: We can ask the ADC to "interrupt" the processor when its done so the processor doesn't have to sit there polling

```
while (ADCSRA & (1 << ADSC));
```
When Exceptions Occur...

- **What should happen?**
  - We could be anywhere in our software program...who knows where

- **Common approach...**
  - Save place in current code
  - Automatically have the processor call some function/subroutine to handle the issue (a.k.a. *Interrupt Service Routine* or *ISR*)
  - Resume normal processing back in original code

```c
#include<avr/io.h>

void codeToHandleInterrupt();

int main()
{
    // this is just generic code
    // for a normal application
    PORTC |= (1 << PC2);
    int cnt = 0;
    while(1){
        if( PINC & (1 << PC2) ) {
            cnt++;
            PORTD = segments[cnt];
        }
    }
    return 0;
}

ISR()
{
    // do something in response to the event
}
```

If an interrupt happens here...

...the processor will automatically call a predetermined function (a.k.a. ISR)

...then resume the code it was executing previously
When Exceptions Occur...

- How does the processor know which function to call "automatically" when an interrupt occurs?
- We must tell the processor in advance which function to associate (i.e., call) with the various exceptions it will check for.
- Just like a waiver forms asks for an emergency contact to call if something bad happens, we indicate what function to call when an interrupt occurs.

```c
#include<avr/io.h>

unsigned char value = 0;
void adcFinished()
{
}

int main()
{
    // this is just generic code
    PORTC |= (1 << PC2);
    ADMUX = 0x61;
    // start first conversion
    ADCCRA |= 0x40;
    while(1)
    {
        /* do useful work here */
        return 0;
    }
}

ISR(ADC_vect)
{
    // ADC is now done
    value = ADCH;
    // output value
    PORTD = value;
    // start next conversion
    ADCCRA |= 0x40;
}
```

If an interrupt happens here...

...the processor will automatically call a predetermined function.
AVR INTERRUPT SOURCES
Interrupt Sources

• An AVR processor like the ATmega328P has numerous sources of possible interrupts
  – External signals
  – Internal modules
  – plus a few others
Interrupt Sources

• Pin Change Interrupt can detect if any pin that is part of a particular PORT (i.e. B, C, D) has changed its value
  – Interrupt if a pin changes state (0→1 or 1→0)
  – 3 individual pin change interrupts
    • Pin Change Interrupt 2 = any bits on PORTD change
    • Pin Change Interrupt 1 = any bits on PORTC change
    • Pin Change Interrupt 0 = any bits on PORTB change
  – Interrupt only says a pin of the port has changed but not which one
    • The function that gets called can figure out what happened by reading the PINx register and AND'ing it appropriately just like we did in previous labs
  – Useful if you have to monitor a number of external sources for changes
More Interrupt Sources

• Counter/Timers
  – Most processors have some hardware counters where you load them with a value, N, and then they automatically count up from 0 to that value at some rate
  – When they reach N they can generate an interrupt and even start over so that an interrupt is generated at a regular interval
  – ATmega328P has three such timers that can be used
  – Useful for performing operations at specific time intervals.
    • Every time an interrupt occurs, do something or measuring how long it takes an event to occur
Other Interrupt Sources

• Communications modules
  – The AVR has several serial communications modules built on (think of these like old-school USB interfaces)
  – Interrupts can be configured to occur when data is received, sent, etc.

• ADC module
  – The ADC can generate an interrupt once it's done converting the voltage (i.e. you start it and then it will "interrupt" you to tell you its done)
External Interrupt Pins

• An 'external' interrupt can be generated anytime a certain external trigger condition occurs on two specific pins
  – INT0 = PIN4 = PORTD2 or
  – INT1 = PIN5 = PORTD3
  – Can select condition to cause interrupt
    • Transition from 0→1
    • Transition from 1→0
    • Either kind of transition
    • 0 level
Enabling Interrupts

• All interrupt sources must be enabled before they can be used.

• Each source of an interrupt has its own interrupt enable bit:
  – Located in one of the registers that controls it for the module.
  – Usually 0 = Don't use interrupts, 1 = Enable use of interrupts.
  – Example: \( \text{ADCSRA} |= (1 << \text{ADIE}); \)

• Processor has a global interrupt enable bit in the status register:
  – \( I - \)bit = 0 \( \Rightarrow \) all interrupts are ignored.
  – \( I - \)bit = 1 \( \Rightarrow \) interrupts are allowed.
  – Set or clear in C with the “sei()” and “cli()” function calls.

• Summary: For a module to generate an interrupt:
  – The global I-bit must be a one.
  – The local interrupt enable bit must be a one.
  – Something must happen to cause the interrupt.
When Interrupts Occur

- When an interrupt occurs several things happen
  - Global I-bit cleared to 0 ⇒ no more interrupts can occur while we handle this one
  - Save the current location in our program
  - Call the code (i.e. the ISR) that handles that interrupt using something called the "interrupt vector table"
Who You Gonna Call?

- The HW maintains a table/array (a.k.a. **interrupt vector table**) in memory
  - Each location in the table is associated with a specific interrupt
  - Each entry specifies which function to call when that interrupt occurs
- When a certain interrupt occurs, the HW automatically looks up the ISR/function to call in the table and then calls it

<table>
<thead>
<tr>
<th>Interrupt that HW Associates with this entry...</th>
<th>Table Entry</th>
<th>...Which User Defined Function to Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>External Interrupt Request 0</td>
<td>1</td>
<td>ISR(INT0_vect)</td>
</tr>
<tr>
<td>External Interrupt Request 1</td>
<td>2</td>
<td>ISR(INT1_vect)</td>
</tr>
<tr>
<td>Pin Change Interrupt Request 0 (Port B)</td>
<td>3</td>
<td>ISR(PCINT0_vect)</td>
</tr>
<tr>
<td>Pin Change Interrupt Request 1 (Port C)</td>
<td>4</td>
<td>ISR(PCINT1_vect)</td>
</tr>
<tr>
<td>Pin Change Interrupt Request 2 (Port D)</td>
<td>5</td>
<td>ISR(PCINT2_vect)</td>
</tr>
<tr>
<td>Watchdog Time-out Interrupt</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Timer/Counter2 Compare Match A</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Timer/Counter0 Compare Match A</td>
<td>14</td>
<td>ISR(TIMER0_COMPA_vect)</td>
</tr>
<tr>
<td>Timer/Counter0 Compare Match B</td>
<td>15</td>
<td>ISR(TIMER0_COMPB_vect)</td>
</tr>
<tr>
<td>Timer/Counter0 Overflow</td>
<td>16</td>
<td>ISR(TIMER0_OVF_vect)</td>
</tr>
<tr>
<td>SPI Serial Transfer Complete</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>USART Rx Complete</td>
<td>18</td>
<td>ISR(USART_RX_vect)</td>
</tr>
<tr>
<td>USART Data Register Empty</td>
<td>19</td>
<td>ISR(USART_UDRE_vect)</td>
</tr>
<tr>
<td>USART Tx Complete</td>
<td>20</td>
<td>ISR(USART_TX_vect)</td>
</tr>
<tr>
<td>ADC Conversion Complete</td>
<td>21</td>
<td>ISR(ADC_vect)</td>
</tr>
<tr>
<td>EEPROM Ready</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>Analog Comparator</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>Two-wire Serial Interface</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>Store Program Memory Read</td>
<td>25</td>
<td></td>
</tr>
</tbody>
</table>
Interrupt Service Routines

- The vector table stores which function (a.k.a. “Interrupt Service Routine” or just ISR) should be called when a particular interrupt occurs.
- An ISR is written like any other function (almost)
  - Must be declared as an ISR for a specific interrupt by using a special name [e.g. `ISR(ADC_vect)`]. Compiler then knows to put the address of the function in vector table
  - No arguments can be passed
  - No values can be returned
- ISRs have access to other functions and global variables like any other function.
  - Danger! Changing global variables from an ISR can cause problems for the rest of the program that’s not expecting things to be changed.
  - Ways to prevent problems (more later.)

```c
#include<avr/io.h>
#include<avr/interrupt.h>

int main()
{
   ...
}

ISR(ADC_vect)
{
   // ADC is now done
   value = ADCH;
   // output value
   PORTD = value;
   // start next conversion
   ADCSRA |= 0x40;
}
```
Interrupt Example

• Example: Arduino ADC conversions
• Doing conversions without interrupts
  – Start conversion
  – Loop checking status bit until done (called “polling”)
  – Read results
  – Works but program is tied up during the conversion process
  – Gets worse if many tasks have to be done simultaneously
• Better to use interrupts
  – Start conversion, and tell ADC to generate an interrupt when done
  – Program now free to do other things
  – When conversion complete, ISR is executed to read the results
  – Program can start several tasks, and handle each when they finish
Interrupt Example

- Polling method:
  - Start the conversion
  - Loop checking to see when the conversion is complete
  - Read result and take action

```c
#include<avr/io.h>

int main()
{
    // Initialization code here
    ADMUX = ??
    ADSCRA = ??

    while (1) {
        // Start a conversion
        ADCSRA |= (1 << ADSC);

        // Wait for conversion complete
        while (ADCSRA & (1 << ADSC));

        // Read result
        n = ADCH;

        // Do something with n
    }
}
```
Interrupt Example

- Interrupt method:
  - `#include <avr/interrupt.h>`
  - Enable interrupts
  - Start the ADC the first time
  - Loop using the results
  - As the ADC finishes it will call the ISR associated with the ADC which will read the data and start the next conversion
- Note the special syntax for how to declare the function associated with the ADC
  - `ISR(ADC_vect)`
  - Always start with `ISR( )` and then has the name of the interrupt vector

```c
#include<avr/io.h>
#include <avr/interrupt.h>

// global variable
int adc_data = 0;

int main()
{
  // Init. code here (ADMUX, etc.)

  // Enable ADC interrupts
  ADCSRA |= (1 << ADIE);
  // Enable global interrupts
  sei();
  // Start first ADC conversion
  ADCSRA |= (1 << ADSC);

  while (1) {
    // Update display using adc_data
    // which will be automatically
    // updated each time interrupt occurs;
  }
}

// Read the conversion results
ISR(ADC_vect)
{
  adc_data = ADCH; // Read result
  // Start next conversion
  ADCSRA |= (1 << ADSC);
}
```
Interrupt Flag Bits

- Modules usually contain an interrupt flag (IF) bit in the same register as the interrupt enable (IE) bits
  - Flag is set when the module wants to generate an interrupt.
  - Flag is cleared when the ISR is called
  - Allows the program to see if interrupts would have occurred if they were enabled (i.e. If we aren't using interrupts for the ADC we can still look at the ADIF bit to see if it would have tried to generate an interrupt)

<table>
<thead>
<tr>
<th>ADEN</th>
<th>ADSC</th>
<th>AD</th>
<th>ATE</th>
<th>ADIF</th>
<th>ADIE</th>
<th>AD</th>
<th>AD</th>
<th>AD</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS2</td>
<td>PS1</td>
<td>PS0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ADCSRA Register
Disclaimer 2

• All processors handle interrupts differently.
  – The AVR is typical in some ways, not in others.
  – If working with a different processor, don’t assume it works the same as the AVR.
  – READ THE MANUAL!
Declaring volatile variables and using atomic operations

DEALING WITH ASYNCHRONOUS EVENTS
Volatile Variables

- Global variables can be shared between main code and ISR’s.
- ISR’s can modify the contents of a global variable.
- Variable can change despite no assignment statement in main program.

```c
main() {
    // Loop waiting for flag non-zero
    while (flag == 0);
    // Do something
}
ISR(SOME_INTERRUPT_vect) {
    flag = 1;
}
```
Volatile Variables

- Example: When optimizing this code, compiler sees that “flag” is never changed inside the loop and it will simplify the main program code.
- From standpoint of the compiler, why check the variable again since it could not have changed?

```c
main() {
    // Loop waiting for flag non-zero
    while (flag == 0);

    // Do something
}
ISR(SOME_INTERRUPT_vect) {
    flag = 1;
}

Result of compiler optimization

main() {
    if (flag == 0)
        while (1);
    else
        // Do something
}
Volatile Variables

- Solution: All global variables that might be modified by the HW directly or by an ISR should be declared as "volatile"
- Tells the compiler the value can change at any time, and must be checked each time used.

```c
volatile char flag;
main() {
    // Loop waiting for flag non-zero
    while (flag == 0);
    // Do something
}
ISR(SOME_INTERRUPT_vect) {
    flag = 1;
}
```
Need for Atomic Operations

- Sometimes performing an operation requires several steps (ex. Copying bits into a register)
- If an interrupt occurs in the middle of the sequence it may see a strange value/state of the variable and do something we didn't expect
- Atomic operations are compound statements that should execute all together (not be interrupted)

```c
#define MASK 0b00001111

main()
{
    PORTD = 0x0f; // PORTD starts at 1's
    char x = 0x05;

    // copy lower 4 bits of x to PORTD
    PORTD &= ~MASK;
    PORTD |= (x & MASK);

    // we would expect PORTD to end w/ 5
    // but what if interrupt occurred
    // between these two lines
}

ISR(SOME_INTERRUPT_vect)
{
    if( (PORTD&MASK) == 0){
        // do something
    }
}
```
Atomic Operations

• “Atomic” ⇒ Can’t be interrupted while executing
• The problem gets worse at the assembly level since many C operations (one line of code) require multiple assembly language instructions, and interrupts can occur between them.
  – All multibyte values (int, long, etc.) are handled in two or more steps.
• Need a way to ensure all operations occur together and are not interrupted (e.g. ensure an interrupt doesn't occur in the middle)
• Solution:
  – Disable interrupts before the operations - cli();
  – Perform the desired compound operations
  – Re-enable interrupts – sei();
Updated Code for Atomicity

• Disable interrupts using cli()
• Perform the operation
• Re-enable interrupts using sei()

```c
#define MASK 0b00001111
main()
{
    PORTD = 0x0f; // PORTD starts at 1's
    char x = 0x05;
    cli();
    // copy lower 4 bits of x to PORTD
    PORTD &= ~MASK;
    PORTD |= (x & MASK);
    sei();

    // now we can't be interrupted
}

ISR(SOME_INTERRUPT_vect)
{
    // if lower 4 bits all = 0
    if( (PORTD&MASK) == 0){
        // do something
    }
}
```
Built-In Atomic Block

- In a larger program there are some issues that might arise
  - OK to disable interrupts, but shouldn’t turn them back on if they were already disabled by some other code
- Solution: Use atomic.h and the ATOMIC_BLOCK()
- Turns interrupts off, then restores to previous state.

```c
#include <util/atomic.h>

main() {

    ATOMIC_BLOCK() {
        \ interrupt now off
        \ Do something
    }\ interrupt setting restored
}

ISR(SOME_INTERRUPT_vect) {
}
```
Exercise

• First finish your ADC lab and demo it!
• Then convert your ADC lab code to use interrupts
  – Make sure you save a copy of your working ADC lab code as a backup