Unit 12

Exceptions & 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

- In computer systems we may NOT know when
  - External hardware events will occur.
    - Can you think of an example?
  - Errors will occur
- Exception processing refers to
  - Handling events whose timing we cannot predict
- 3 questions to answer:
  - Q: Who detects these events and how? A: The hardware
  - Q: How do we respond? A: Calling a pre-defined SW function using the vector table
  - Q: What is the set of possible events? A: Specific to each proc.

An Analogy

- Scenario:
  - You're studying (i.e. listening to music and watching Netflix) but all of a sudden you get a text message. What do you do?
  - You stop what your doing and message back
  - When you're done you go back to studying (i.e. playing a video game or going to get coffee)
- This is what computers do when an ______________________ occurs

What are Exceptions?

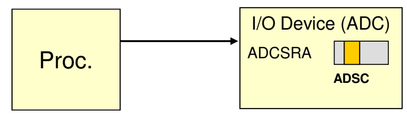
- **Definition**: Any event that causes a ________________ ______________________
  - "Exceptions" is a broad term to catch many kinds of events that *interrupt* normal software execution
- Examples
  - **Hardware Interrupts / __________ Events [Focus for today]**
    - PC: Handling a keyboard press, 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
  - _____________________ (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
  - ______________ (responsibility on I/O device)
    - I/O device notifies processor only when it needs attention

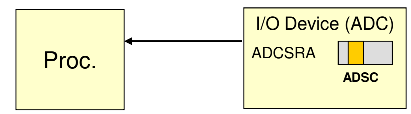
Recall: Once the A-to-D converter has started we need to wait until the ADSC bit is 0 (i.e. keep waiting 'while' ADSC bit is 1)

```
while (ADCSRA & (1 << ADSC));
```



Polling Loop

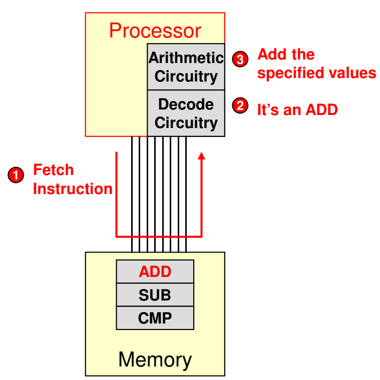
With Interrupts: We can ask the ADC to "interrupt" the processor when its done so the processor doesn't have to sit there polling



Interrupt

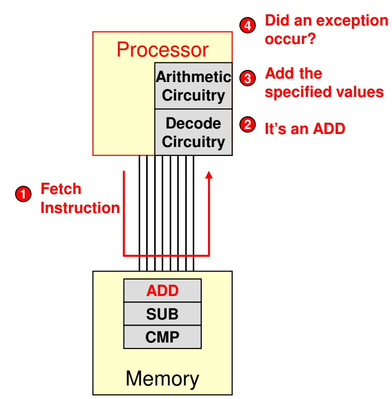
Recall: Instruction Cycle

- Processor hardware 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**



HW Detects Exceptions

- There's actually a 4th step
- After finishing each instruction the processor hardware checks for ____________________ automatically (i.e. this is built into the hardware)
  - 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



SW Handles Exceptions

- When exceptions occur, what should happen?
  - We could be anywhere in our software program…who knows where
- Common approach…
  - 1. __________ in current code
  - 2. Automatically have the processor call some function/subroutine to handle the issue (a.k.a. __________________ or _____)
  - 3. Resume normal processing back in original code

```
#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

Important Point:
HW detects exceptions.
Software handles exceptions.

When Exceptions Occur…

- How does the processor know which function to call "automatically" when an interrupt occurs
- We must tell the processor in ___________ 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

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

unsigned char value = 0;
void adcFinished();

int main()
{
   // this is just generic code
   PORTC |= (1 << PC2);
   ADCMUX = 0x61;
   // start first conversion
   ADCSRA |= 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
  ADCSRA |= 0x40;
}
```

If an interrupt happens here…

…the processor will automatically call a predetermined function

Function Calls vs. Interrupts

Normal function calls

- ______________: Called whenever the program reaches that point in the code
- Programmer can pass arguments and receive return values

Interrupts

- ____________: Called whenever an event occurs (can be anywhere in our program when the ISR needs to be called)
  - Requires us to know in advance which ISR to call for each possible exception/interrupt
  - Use ISR(*interrupt_type*) naming scheme in the Arduino to make this association
- No ___________ or _______ values
  - How would we know what to pass if we don't know when it will occur
  - Generally interrupts update some global variables

AVR INTERRUPT SOURCES

Interrupt Sources

- An AVR processor like the ATmega328P has numerous sources of possible interrupts
- Some interrupts are from the on-board modules
  - Pin change interrupts
  - Timers/Counters
  - Serial communications
  - A-to-D converter
- A few can be generated from external devices
  - External pin interrupts so you can connect a whole new HW device to your Arduino and have it generate an interrupt

Pin Change Interrupts

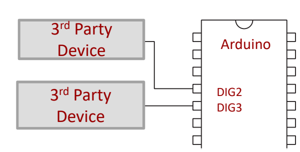
- Pin Change Interrupt can detect if _______ that is part of a particular PORT (i.e. B, C, D) has changed its value
  - Interrupt if a pin changes state (0→1 or 1→0)
  - 3 individual pin change interrupts
    - Pin Change Interrupt 0 = any bits on PORTB change
    - Pin Change Interrupt 1 = any bits on PORTC change
    - Pin Change Interrupt 2 = any bits on PORTD change
  - Interrupt only says ____ pin of the port has changed but not _________ 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

Counter/Timer Interrupts

- Most processors have some hardware counters that count at some ___________________ that you can load with a value, N.
- Start at 0 counting at the known rate and when they reach __ they can generate an _____________
  - If N = 500 and the timer counts at 1KHz, an interrupt will occur after 0.5s
- Can be set to immediately ____________ at 0 again to generate an interrupt at a regular ___________
- ATmega328P has **three** such timers that can be used
- Useful for performing operations at specific time intervals.

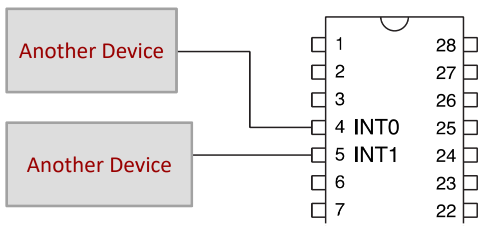
Other Interrupt Sources

- Communications modules
  - The AVR has several serial communications modules built in (think of these like forerunners of modern 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 Interrupts
  - Can be used to connect 3rd party devices to the system and have them generate interrupts on 0->1 or 1->0 transitions
    - PORTD bit 2 and 3



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 ____
    - Transition from ____
    - Either kind of transition
    - 0 level



Who You Gonna Call?

- The HW maintains a table/array (a.k.a. **interrupt vector table**) in memory
  - Each location in the table is ___________ 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
- More on this in your OS class (CS 350) or Architecture course (EE 457)

Page 58 on http://www.atmel.com/images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet_Complete.pdf

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

Interrupt Service Routines

- 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)**]. This tells the compiler to fill in the **interrupt vector table** to call this code when an ADC interrupt occurs.
  - No arguments can be passed
  - No values can be returned
- ISRs have access to other functions and global variables like any other function

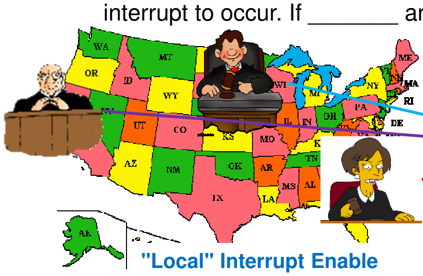
```
#include<avr/io.h>
#include<avr/interrupt.h>

int main()
{
  ...
}

ISR(ADC_vect)
{
  // ADC is now done
  value = ADCH;
  // output value
  PORTD = value;
  // start next conversion
  ADCSRA |= (1 << ADSC);
}
```


To Use Arduino Interrupts

- Define the ______ in your software program
- During initialization you must _________ the interrupt source
- Either…
  - Wait for the interrupt to occur (e.g. wait for a pin to change)
  - or invoke behavior that will eventually lead to an interrupt (start an ADC conversion so that eventually it will generate an interrupt when done)

Enabling Interrupts

- Each interrupt source is DISABLED by default and must be ENABLED
- For an interrupt to be handled, ______ "enablers" need to agree
  - **Enabler 1**: A separate "________" interrupt enable bit per source (i.e. ADC, timer, pin change, etc.)
  - **Enabler 2**: A single "________" interrupt enable bit (1-bit for entire processor called the I-bit)
- Analogy: Local judge per state but 1 supreme court for entire nation
  - Both local judge and supreme court must agree (be set to '1') for the interrupt to occur. If _______ are '0' then the interrupt will ______ occur.



"Local" Interrupt Enable
(1 per interrupt source)

"Global" Interrupt Enable
(1 for entire system)

Enabling Interrupts

- All interrupt sources must be enabled before they can be used
- Each source of an interrupt has its own _______________ bit
  - Located in one of the registers that controls it for the module
  - Usually __ = Don't use interrupts, ___ = Enable use of interrupts
  - Example: `ADCSRA |= (1 << ADIE);`

ADEN	ADSC	AD ATE	ADIF	ADIE	AD PS2	AD PS1	AD PS0

- Processor has a _____________ enable bit in the status register
  - I-bit = 0 ⇒ all interrupts are ignored
  - I-bit = 1 ⇒ 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

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

```
#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
- Be sure to follow 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
- **Note**: If you enable an interrupt but have no ISR() written the Arduino will reboot immediately when the interrupt occurs

```
#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 (Skip)

- 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)

ADEN	ADSC	AD ATE	ADIF	ADIE	AD PS2	AD PS1	AD PS0

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!

NEED FOR VOLATILE VARIABLES

Communicating with ISRs & Other Code

- Global variables can be shared between main code and ISR's
- ISR's can modify the contents of a global variable and other code and check for changes in that global variable
- Common idiom: a "_______" variable to indicate a desired event has happened
  - ISR ________ on every change to see if the desired event occurred and only then _____ a flag to __
  - Main or other code checks the flag variable then ______ it to __

```
int flag; // shared variable
main()
{
  flag = 0;

  // Loop waiting for interrupt to occur
  while (flag == 0);

  // Do something
}

ISR(SOME_INTERRUPT_vect)
{
  flag = 1;
}
```


Using a Flag Variable

- A common idiom is to use a "flag" variable to indicate a desired event has happened
- Approach
  - Initialize a global variable to 0
  - ISR checks on every change to see if the desired event occurred and only then sets a flag to 1
  - Main or other code checks the flag variable then resets it to 0 awaiting the next time the event occurs

```
int pb2flag; // shared variable

main() {
  pb2flag = 0;  // Initialize to 0
  while(1){
    // Loop waiting for flag to be set
    if (pb2flag == 1){
       stringout("button push!");
       pb2flag = 0; // reset flag to 0 so we
                    // can detect the next push
    }
    // Check for other things or do work
  }
}
ISR(PCINT0_vect)
{
  // Some bit changed, see if it is PB2
  if( (PINB & PB2) == 0){
    pb2flag = 1;
  }
}
```


ISR Timing

- Why not just do the work in the ISR?
- Because an _________ can not ____________ another _____________!
  - That's a mouthful
  - When you are in an ISR no other interrupts can occur possibly delaying important events, or even ________ information (e.g. a "high-speed" communications link with limited space)
- **Solution**: Never do _________ __________ work in an ISR

```
main() {
  while(1){
    // Check for other things or do work
  }
}
ISR(PCINT0_vect)
{
  // Some bit changed, see if it is PB2
  if( (PINB & PB2) == 0){
    stringout("button push!");
  }
}
```

Main Point:
Get __________ of an ISR quickly.

Don't call functions that will take a long time to complete (e.g. LCD output)

Another Issue: Compiler Optimizations

- Example: When optimizing this code, compiler sees that "flag" is never modified in main (and doesn't see any "calls" to the ISR)
- Thus the compiler will optimize the code to avoid reading "flag" from memory each time (since that is slow)
- Problem: Due to the compiler optimization our code won't work even if the ISR sets the flag to 1
- Solution: Tell the compiler that "flag" can change due to some ISR by declaring it as ***volatile***

Original Code

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

    // Do something
}

ISR(SOME_INTERRUPT_vect)
{
    flag = 1;
}
```

If you just look at main, would you expect this while loop to ever be false?

Result of compiler optimization

```
int flag;
main() {
    flag = 0;
    // compiler optimized result
    while (______);

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

Another Issue: Compiler Optimizations

- **Solution**: Tell the compiler that "flag" can change due to some ISR by declaring it as ***volatile***
  - Declaring a global variable as volatile tells the compiler not to optimize the code but always get the __________ value of the variable
- **Important Rule:** Use "volatile" for any global variable that is updated in an ISR and used elsewhere in the code
- **Corollary**: No need to use "volatile" for variables not used with ISRs (e.g. "buf" in the example at the right).

Original Code

```
char buf[17]; // not used in an ISR.
volatile int flag;
main() {
    flag = 0;
    // Loop waiting for flag non-zero
    while (flag == 0);

    // Do something
    snprintf(buf,3,"Hi");
}

ISR(SOME_INTERRUPT_vect)
{
    flag = 1;
}
```

Volatile declaration tells compiler to always look at the latest value of "flag"

Performing critical sections without be interrupted

NEED FOR ATOMIC OPERATIONS

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)

```
#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);
   // both lines should be done "together"

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

}

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


Atomic Operations

- "Atomic" ⇒ Can't be ______________ 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.
  - Even "x++" is actually 3 steps: ______ old value of x, ____ 1, ________ x to new value
- Need a way to ensure all operations occur __________ and are not interrupted (e.g. ensure an interrupt doesn't occur in the middle)

Updated Code for Atomicity

- Solution to ensure "atomic" operation
  - Disable interrupts using cli()
  - Perform the operation
  - Re-enable interrupts using sei()
- The **code** between **cli()** and **sei()** that cannot be interrupted is called a **"critical section"** (since it must be done together)

```
#define MASK 0b00001111
main() {

   PORTD = 0x0f; // PORTD starts at 1's
   char x = 0x05;

   cli();
   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
    }
}
```

Code between cli() and sei() is called a "critical section"

Can't happen during the critical section

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.

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

main() {

    ATOMIC_BLOCK()
    { /* like cli() */
        // interrupts now off
        // Do critical section
    } /* like sei() */

    // interrupt setting restored
}

ISR(SOME_INTERRUPT_vect)
{
}
```