Interrupts
Arduino, AVR, and deep dark programming secrets

What is an Interrupt?
- A transfer of program control that is not directed by the programmer
- Like a phone call in the middle of a conversation
- Stop what you are doing, deal with the interruption, then continue where you left off
- Very handy for handling events that need immediate attention
  - Or that need to occur at regular intervals
  - Or that need to run automatically without the programmer keeping track

What Happens
- An interrupt is signaled somehow
  - A phone rings
- The AVR stops running user code and checks to see what caused the interrupt
  - Stop your conversation and check which phone is ringing
- The AVR runs an Interrupt Service Routing (ISR) related to that interrupt
  - Answer the phone and handle the call
- The AVR restores the system state and picks up the user code where it left off
  - Hang up and resume your previous conversation

Types of Interrupts
- On Arduino/AVR, there are three types
  - External: A signal outside the chip (connected to a pin)
  - Timer: Internal to the chip, like an alarm clock
  - Device: One of the AVR devices (USART, SPI, ADC, EEPROM) signals that it needs attention

Example: USART
- USART handles the serial communication between Arduino and the host
  - Why not just check for a new character in a loop?
  - How frequently would you have to check?
  - How much processor time would be spent checking?

Example: USART
- Serial port at 9600 baud (9600 bits/sec)
  - Each bit is sent at 9.6 kHz (close to 10kHz)
  - Each bit takes around 100usec
  - Around 10 bits required for each character
  - So, one character every 1ms or so
  - If the USART is buffered, you have about 1ms to get a character before it's overwritten by the next one
  - So, you need to check faster than once every millisecond to keep up (around 1000 times a sec)
  - If your main loop is not doing anything else, you can do this, but if you're doing other things, or communicating at faster speeds, it gets ugly fast
Example: USART

- Instead of setting up an interrupt handler for the USART
- The USART will cause an interrupt each time it receives a complete character
- The Interrupt Service Routine (ISR) for this USART receive event will be called
- The ISR will take the character from the USART and put it in a buffer for your program to use
- You never have to check the USART directly, characters just show up in your program’s buffer as they arrive

Types of Interrupts

- On Arduino/AVR, there are three types
  - External: A signal outside the chip (connected to a pin)
  - Timer: Internal to the chip, like an alarm clock
  - Device: One of the AVR devices (USART, SPI, ADC, EEPROM) signals that it needs attention

External Interrupts

- An external event (signal on an input pin) causes an interrupt
- A button, a sensor, an external chip, etc.
- There are two external interrupt pins on Arduino
  - Interrupt 0 (Pin 2) and Interrupt 1 (Pin 3)
- Supported by the Arduino software
  - attachInterrupt(interrupt#, func-name, mode);
  - Interrupt# is 0 or 1
  - Func-name is the name of the ISR function
  - Mode is LOW, CHANGE, RISING, or FALLING

From the Arduino Reference

Using interrupts

Interrupts are useful for making things happen automatically in microcontroller programs, and can help solve timing problems. A usual task for using an interrupt might be reading a sensor module, monitoring user input.

If you wanted to make a program always check the pulse from a sensor module, never missing a pulse, it would make it very tricky to write a program to do anything else, because the program would need to constantly poll the sensor for its data, in order to catch pulses when they occurred. Other sensors have a similar situation element too, such as trying to monitor an analog voltage over time or trying to avoid collisions in motion. In all of these situations, using an interrupt can free the microcontroller so that other work done while not missing the data.

- Two other Arduino functions:
  - interrupts();  // enables interrupts
  - sei();  // enables interrupts (AVR)
  - noInterrupts();  // disables interrupts
  - cli();  // disables interrupts (AVR)

Reference: Language oriented | Libraries | Comparison | Changes
attachInterrupt(interrupt#, func-name, mode);
Description
On Arduino, the function is called when an external interrupt occurs. Specify any function that was attached to the interrupt.

Parameters:
- interrupt#: the number of the interrupt pin
- func-name: the function to call when the interrupt occurs; this function must take no parameters and return nothing. This function is sometimes referred to as an interrupt service routine.
- mode: defines when the interrupt should be triggered; four constants are predefined as valid values:
  - LOW: to trigger the interrupt whenever the pin goes low
  - CHANGE: to trigger the interrupt whenever the pin changes value
  - RISING: to trigger when the pin goes from low to high
  - FALLING: to trigger when the pin goes from high to low

Returns:
- void

Note
Inside the attached function, delay() won’t work and the value returned by read() will not increment. Ensure data review while in the function may be lost. You should declare all variables that you modify within the attached function.

External Interrupt Example

```c
int pin = 13;  // the builtin LED pin
volatile int state = LOW;  // Hold the state of the LED

// Note that external interrupt 0 looks for changes on
// digital pin 2 of the Arduino board
void setup(){
  pinMode(pin, OUTPUT);
  attachInterrupt(0, blink, CHANGE);  // attach ISR
}

void loop(){
  digitalWrite(pin, state);  // Main code writes to LED
}

void blink() { state = !state;}  // ISR changes LED state
```
Aside: Volatile Qualifier

volatile keyword

volatile is a keyword known as a volatile qualifier. It is usually used before the datatype of a variable, to modify the way in which the compiler and subsequent program treats the variable.

Receive a variable volatile is a directive to the compiler. The compiler is software which translates your C/C++ code into the machine code, which are the real instructions for the machine chips in the include.

Typically, it directs the compiler to load the variable from RAM and not from a storage register, which is a temporary memory location where program variables are stored and manipulated. Under certain conditions, this value for a variable stored in registers will be indeterminate.

A variable should be declared volatile whenever its value can be changed by something beyond the control of the code section in which it appears, such as a concurrently executing thread. In the Arduino, the only place that this is likely to occur is in sections of code associated with interrupts, called as interrupt service routines.

Another External Interrupt Example

Generally, the interrupt hardware calls this when the hit our bumper

```
void bumperISR() {
  Stop();  // stop forward motion
  bumper = 1;  // indicate that the bumper was hit
  DriveBackward();  // set motors to reverse
  delay(1000);  // back up for 1 second
  TurnRight();  // turn right (away from obstacle)
  DriveForward();  // drive off again...
}
```

```
void loop() {
  // You can put any other robot driving commands here
  // but you don’t need to check for the bumper here.
  // It’s handled by the external interrupt.

  // If you want to, you can check the value of the bumper
  // variable in the main code to see if it was hit. If you do
  // check, you can reset it to 0 so that you can continue to
  // check later.
}
```

Aside – more external interrupts

- Arduino (AVR) has only 2 external interrupt pins
- Actually, if you want CHANGE mode, there are lots more pins you can use (pretty much all the Arduino pins)
- But, that requires a little deep dark secret AVR-hacking
- So, unless you need it, don’t worry about it
- If you do need it – Look at the PC Int code on the Arduino site
  - Original code that allows triggering an interrupt from any pin on the Arduino...
  - I’ll put a link on the class web site

Another External Interrupt Example

- Interrupt-Driver Bumper Example for a robot
- A bumper switch on the front of the robot should be tied to digital pin 2 and ground

```
#include <interrupt.h>

volatile int bumper;  // Indicator to the main code have we hit something

void setup() {
  pinMode(2, INPUT);  // Make digital 2 an input (for the bumper switch)
  digitalWrite(2, HIGH);  // Enable pull up resistor (bumper switch pulls low)
  attachInterrupt(0, bumperISR, FALLING);  // attach our interrupt pin (pin 2) to its ISR

  interrupts();  // interrupts are enabled by default, but this doesn’t hurt

  // start moving
  bumper = 0;
  DriveForward();
}
```

```
void bumperISR() {
  Stop();  // stop forward motion
  bumper = 1;  // indicate that the bumper was hit
  DriveBackward();  // set motors to reverse
  delay(1000);  // back up for 1 second
  TurnRight();  // turn right (away from obstacle)
  DriveForward();  // drive off again...
}
```

External Interrupt Summary

- AVR ATmega328p has 2 external interrupts
  - 0 (on Arduino pin 2) and 1 (on Arduino pin 3)
  - Use attachInterrupt(int#, ISR-name, mode); to attach an ISR to an external interrupt
  - Make sure to provide a function definition for ISR-name
  - Choose mode as LOW, CHANGE, RISING, FALLING
  - If the main code looks at a variable that is set in the ISR, make sure that variable is volatile
  - detachInterrupt(int#); is also available
  - interrupts(); and noInterrupts(); turn them on and off

Types of Interrupts

- On Arduino/AVR, there are three types
  - External: A signal outside the chip (connected to a pin)
  - Timer: Internal to the chip, like an alarm clock
  - Device: One of the AVR devices (USART, SPI, ADC, EEPROM) signals that it needs attention
Motivation

Arduino 101 – blinky LED

Problem – Arduino is just wasting time during the delay. It can’t be used for anything else.

```c
int ledPin = 13; // LED connected to digital pin 13
void setup() {
    pinMode(ledPin, OUTPUT); // initialize the digital pin as an output.
}
void loop() {
    digitalWrite(ledPin, HIGH); // set the LED on
    delay(1000); // wait for a second
    digitalWrite(ledPin, LOW); // set the LED off
    delay(1000); // wait for a second
}
```

Non-delay version – use a timer to see if it’s time to blink

Can use the Arduino for other things in the meantime

But, the programmer has to manage this activity

Don’t use delay – that ties up the processor while it’s delaying

Instead, there is a `millis`() function that returns the current number of milliseconds since the last system reset

Based on internal timers!

Use that to check occasionally if enough time has passed that you should flip the LED again

You can do other things between checking...
AVR Timers

- Our Arduino’s AVR has three internal timers
- Timer0: an 8-bit timer (counts 0 to 255)
  - Used for system timing, millis(), micros(), etc.
  - and PWM on pins 5 and 6
- Timer1: a 16-bit timer (counts 0 to 65,535)
  - Used for PWM on pins 9 and 10
- Timer 2: an 8-bit timer (counts 0 to 255)
  - Used for PWM on pins 3 and 11

- Don’t use Timer0 – it will mess things up…
- If you use Timer1 or Timer2, you will lose PWM on some pins…

Timer Normal Mode

- Start counting on system reset
- Count until you get to your TOP, then start again at 0
- 8-bit timer TOP is 255
- 16-bit timer TOP is 65,535
- Access a timer’s current value using a special register
  - TCNT0, TCNT1, TCNT2
  - A = TCNT2; // Read the value of timer 2
  - TCNT1 = 0; // Reset the value of timer 1 to 0

How Fast to Count?

- 16MHz is fast!
- 16,000,000 ticks/sec, 62.5ns per clock tick
- A "prescaler" slows down the rate at which a timer counts by some factor
- Increases the range of time you can count, but makes the smallest tick resolution larger
- Timer0 and Timer1: divide clock by 1, 8, 64, 256, 1024
- Timer2: divide clock by 1, 8, 32, 128, 256, 1024

Resolution/Timing with Prescaler

Pre-scale Value | Tick Time | OVF Frequency | OVF Period
--- | --- | --- | ---
1 | 62.5ns | 62.5kHz | 16µsec
8 | 500ns | 7812.5kHz | 128µsec
64 | 4µsec | 1.953125kHz | 512µsec
256 | 1µsec | 39062.5Hz | 5.12µsec
1024 | 64µsec | 156.25Hz | 16.384ms

Resolution/Timing with Prescaler

8-bit counter at 16MHz system clock frequency (Timer2)
OVF = Overflow (time it takes to count from 0 to TOP)
TOP = 255 for an 8-bit counter

Resolution/Timing with Prescaler

Pre-scale Value | Tick Time | OVF Frequency | OVF Period
--- | --- | --- | ---
1 | 62.5ns | 62.5kHz | 16µsec
8 | 500ns | 7812.5kHz | 128µsec
64 | 4µsec | 1.953125kHz | 512µsec
256 | 1µsec | 39062.5Hz | 5.12µsec
1024 | 64µsec | 156.25Hz | 16.384ms

Resolution/Timing with Prescaler

16-bit counter at 16MHz system clock frequency (Timer1)
OVF = Overflow (time it takes to count from 0 to TOP)
TOP = 16,535 for a 16-bit counter

Example: Flash LED at 1Hz

- Find a counter prescale that lets us count slowly enough that we can count to 1sec
- Figure out what to count to to get to 1sec
  - timer_count = ((16,000,000/prescale/target_freq) - 1
- Set up counter with the right prescaler, then check if the count is up to timer_count
- Flash the LED and reset the timer to 0
- (16,000,000Hz/1024)/1Hz - 1 = 15,624
- So, if you count 0 to 15,624 at a 1024 prescale, that’s (15,624x64µsec)/16,000,000 = 1sec
Flash LED at 1Hz

```
int LEDPin = 13;  // Built-in LED pin
void setup () {
  pinMode(LEDPin, OUTPUT);  // Make sure it's an output
  // set up timer1 (16-bit timer) in normal up-counting mode
  // set up timer1 (16-bit timer) for prescale of 1024
}
void loop () {
  if (TCNT1 >= 15624) {  // reached 1sec on timer1
    digitalWrite(LEDPin, !digitalRead(LEDPin));  // toggle LEDPin
    TCNT1 = 0;  // reset counter to 0
  }
}
```

Aside: toggle-tweaking

```
digitalWrite(LEDPin, !digitalRead(LEDPin));
```

Easy...
```
digitalWrite(LEDPin, !digitalRead(LEDPin));
boolean FlipFlop = 0;
...
digitalWrite(LEDPin, FlipFlop);
FlipFlop = !FlipFlop;
```

Faster...

How to Configure Timers?

- Set values into internal timer control registers
- TCNTn is the timer count value
- TCCRnA and TCCRnB are the timer control registers
- Each bit in these control registers controls some aspect of the timer’s behavior

**TCCR1A** – Timer/Counter Control Register A

**TCCR1B** – Timer/Counter Control Register B

Detour: Setting bits inside bytes

- TCCR1B is an 8-bit byte
- Want to set bits 2 (CS12) and 0 (CS10) to 1, leave others 0
```
TCCR1B = B00000101;  // overwrite whole byte
TCCR1B = TCCR1B | B00000101;  // leave other bits unchanged
TCCR1B = B00000101;  // shorthand version of above
CS12 = 2 and CS10 = 0 – these are magically set in an included header file
```
Detour: Setting Bits

| = logical OR
| 00101101 | 01100011 = 01101111
| If there’s a 1 in A or B, there’s a 1 in C.
| (1<<CS12)
| CS12 is defined to be 2 (in a secret included file)
| (1<<1) is 1 shifted two places to the left in the byte
| this is 00000100
| CS10 = 0
| So (1<<CS12) | (1<<10) = 00000100 | 00000001
| This equals 00000101
| BV(CS12) = bit(CS12) = (1<<CS12) = 00000100

Flash LED at 1Hz

Flash LED at 1Hz

Flash LED at 1/minute

CTC Mode

Flash LED at 1/minute

CTC Mode

Flash LED at 1Hz

CTC Mode

CTC Mode
Setting CTC Mode

Checking CTC Result

Flash LED at 1Hz (using CTC)

void setup () {
  pinMode(LEDpin, OUTPUT);  // Make sure it's an output
  TCCR1B |= _BV(WGM12);   // Configure timer 1 for CTC mode
  TCCR1B |= (1<<CS12) | (1<<CS10); // Timer1 prescale of 1024
  OCR1A = 15624; // Set CTC compare value in OCR1A register
}

void loop (){
  if (TIFR1 & _BV(OCF1A)) { // reached 15624 - CTC sets the flag
    digitalWrite(LEDpin, !digitalRead(LEDpin)); // toggle LED pin
    TIFR1 = _BV(OCF1A); // reset flag by writing a 1 to it
    // Strange but true…
  }
}

Flash LED at 1Hz (using CTC)

void setup () {
  pinMode(LEDpin, OUTPUT);  // Make sure it's an output
  bitSet(TCCR1B, WGM12); // Configure timer 1 for CTC mode
  bitSet(TCCR1B, CS12); // Timer1 prescale of 1024
  OCR1A = 15624; // Set CTC compare value in OCR1A register
}

void loop (){
  if (bit_is_set(TIFR1, OCF1A)) { // reached 15624 - CTC sets the flag
    digitalWrite(LEDpin, !digitalRead(LEDpin)); // toggle LED pin
    TIFR1 = _BV(OCF1A); // reset flag by writing a 1 to it
    // Strange but true…
  }
}

Details

(TIFR1 & _BV(TIFRA)) & is logical AND
11010110 & 00010001 = 00010000
Only if there's a 1 in A and B is there a 1 in C
Remember _BV(TIFRA) is also (1<<OCF1A)
OCF1A = 1 (all bits are numbered in the magic included file
This ANDs the TIFR1 register with 00000010
Answer is 1 only if the OCF1A bit is 1
Picks off the OCF1A bit

OK – Add Interrupts!

Can configure things so that an interrupt is signaled whenever the CTC compare target is reached
Set interrupt in TIMSKn timer interrupt mask register
Interrupt flag is in TIFRn timer interrupt flag register
ISR(TIMER1_COMPA_vect) {…} // ISR definition
Can also configure for interrupt on overflow
i.e. counting all the way to TOP
ISR(TIMER2_OVF_vect) {…} // ISR definition
TIMSKn and TIFRn (1 and 2)

Flash LED at 1Hz (with Interrupts)

```c
// Arduino-ized version
void setup () {
  pinMode(LEDPin, OUTPUT);  // Make sure it's an output
  bitSet(TCCR1B, WGM12);  // Configure timer for CTC mode
  bitSet(TIFR1, OCIE1A); // enable CTC interrupt on OCR1A compare
  OCR1A = 15624; // Set CTC compare value in OCR1A register
  interrupts();  // Make sure interrupts are enabled (default?)
}

void loop () {
  // Anything else you want to do...
}

ISR(TIMER1_COMPA_vect){
  digitalWrite(LEDPin, !digitalRead(LEDPin)); // toggle LEDPin
}
```

Flash at 1Hz with OVF Interrupt

1. Find a prescale value for a timer that results in a 1Hz overflow rate
2. Configure the timer
3. Set the interrupt to the overflow interrupt
4. Set up the ISR

Resolution/Timing with Prescaler

<table>
<thead>
<tr>
<th>Prescale Value</th>
<th>Tick Time</th>
<th>OVF Frequency</th>
<th>OVF Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>62.5usec</td>
<td>244.14Hz</td>
<td>4.096msec</td>
</tr>
<tr>
<td>8</td>
<td>50usec</td>
<td>30.52Hz</td>
<td>32.768msec</td>
</tr>
<tr>
<td>64</td>
<td>4usec</td>
<td>3.815Hz</td>
<td>262.144msec</td>
</tr>
<tr>
<td>256</td>
<td>16usec</td>
<td>0.954Hz</td>
<td>1.05sec</td>
</tr>
<tr>
<td>1024</td>
<td>64usec</td>
<td>0.238Hz</td>
<td>4.19sec</td>
</tr>
</tbody>
</table>

16-bit counter at 16MHz system clock frequency (Timer1)

OVF = Overflow (time it takes to count from 0 to TOP)

Flash LED at 1Hz (with OVF interrupt)

```c
// Arduino-ized version
void setup () {
  pinMode(LEDPin, OUTPUT);  // Make sure it's an output
  bitSet(TCCR1B, WGM12);  // Configure timer for normal mode (default)
  bitSet(TIFR1, TOIE1); // enable OVF interrupt on Timer1
  OCR1A = 15624; // Set Timer1 prescale of 1024
  interrupts();  // Make sure interrupts are enabled (default?)
}

void loop () {
  // Anything else you want to do...
}

ISR(TIMER1_OVF_vect){
  digitalWrite(LEDPin, !digitalRead(LEDPin)); // toggle LEDPin
}
```
Flash LED at 1Hz (Timer2)

- Use Timer2 (8-bit)
- Overflows at 61Hz
- Count up to 61 overflows to be 1Hz
- Interrupt each time you overflow

Resolution/Timing with Prescaler

<table>
<thead>
<tr>
<th>Prescale Value</th>
<th>Tick Time</th>
<th>OVF Frequency</th>
<th>OVF Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>62.5nsec</td>
<td>62.5kHz</td>
<td>6.25usec</td>
</tr>
<tr>
<td>8</td>
<td>7.8125kHz</td>
<td></td>
<td>50usec</td>
</tr>
<tr>
<td>32</td>
<td>1.9312kHz</td>
<td></td>
<td>2048usec</td>
</tr>
<tr>
<td>64</td>
<td>976.5625Hz</td>
<td></td>
<td>1024usec</td>
</tr>
<tr>
<td>128</td>
<td>1953.125Hz</td>
<td></td>
<td>2048usec</td>
</tr>
<tr>
<td>256</td>
<td>4960.3125Hz</td>
<td></td>
<td>4096usec</td>
</tr>
<tr>
<td>1024</td>
<td>15625Hz</td>
<td></td>
<td>16384usec</td>
</tr>
</tbody>
</table>

8-bit counter at 16MHz system clock frequency (Timer2) OVF = Overflow (time it takes to count from 0 to TOP)

Timer Summary

- Three timers on AVR (You need to read AVR doc for details!!!)
- Timer0 (8-bit) - used for system stuff (PWM on pins 5, 6)
- Timer1 (16-bit) - (PWM on 9, 10)
- Timer2 (8-bit) - (PWM on 3, 11)
- Set and read timer values (counts) in TCNTn register
- Set timer prescale in the TCCRn register
- Tables tell you what the tick-rate and OVF period are for each prescale
- Set timer mode (normal, CTC, PWM (not covered here!) in the TCCRnA and TCCRnB registers
- Change timer TOP value if in CTC mode
- Set new TOP in OCRnA or OCRnB register

Timer Interrupts Summary

- Set interrupt enable in TIMSKn register
- OVF interrupt is TICEn
- CTC interrupts are OCIEnA and OCIEnB
- Interrupt flags are in TIFRn register
- OVF flag is TOVn
- CTC flags are OCFinA and OCFinB
- Set ISR
  - ISR(TIMERn_OVF_vect)
  - ISR(TIMERn_COMPA_vect)
  - ISR(TIMERn_COMPB_vect)

Bit setting and checking

- Set and clear bits in AVR registers
  - sfr = Special Function register (i.e. TCCR1B...)
  - bitSet(sfr, bit)
  - bitClear(sfr, bit)
- Check on a bit in an AVR register
  - bit_is_set(sfr, bit)
  - bit_is_clear(sfr, bit)
- Loop on checking bits (wait for a bit)
  - loop_until_bit_is_set(sfr, bit)
  - loop_until_bit_is_clear(sfr, bit)
Types of Interrupts

- **External**: A signal outside the chip (connected to a pin)
  - Use `attachInterrupt(int #, ISR-name, mode);
  - also `detachInterrupt(int #);
- **Timer**: Internal to the chip, like an alarm clock
  - Set timer features (normal, CTC, etc.)
  - Set compare values if needed (new TOP)
  - Set interrupt enables (OVF, CTC)
  - Set ISR
- **Device**: One of the AVR devices (USART, SPI, ADC, EEPROM) signals that it needs attention
  - Probably don’t want to mess with these… Arduino does the right thing (but check AVR doc for details)
  - i.e. `analogRead` uses ADC, `spi_write` uses SPI, `println` uses USART, etc.

PWM?

- **PWM also uses timers**
- You can set the timers so that they automatically toggle a pin
  - Specifically - there are two pins assigned to each timer
  - That’s why each timer does two pins worth of PWM on Arduino
    - Timer0 uses pins 5,6, Timer1 uses pins 9,10, Timer2 uses pins 3,11
  - Control the toggle speed with the timer prescale, or with the CTC timer compare
    - That’s what `analogWrite` does – change the OCRnA or OCRnB value to change the CTC compare value
  - Two types of PWM – “fast” and “phase-correct”
    - Subtle difference, but phase-correct has a max speed that’s half as fast as “fast”

Final Word

- Interrupts are a wonderful way of reacting to events, or setting things up to happen at specific times or frequencies
  - Once they’re set up, they operate on their own without main-program fussing
- You can also write wonderfully incomprehensible code that uses interrupts!