**Arduino interrupt programming notes:**

How to read multiple RC channels:

Attach a “pin change interrupt” to the signal pin.

Create an “interrupt service routine” which will be called whenever the signal pin changes from high to low or low to high.

In the interrupt service routine, we: check if signal pin is high or low. If high, this is the rising edge of the pulse, record the time using “micros()” a method to get current time. If low, this is the falling edge. We are interested in pulse duration, so we call micros() and subtract it to get the pulse duration.

**Volatile Shared Variables**: When you upload a program with Arduino IDE, your code becomes an executable that can be run on the Arduino. We take care to avoid unexpected results during compiling (with interrupts).

When we use the ‘volatile’ keyword, it tells the compiler never to store the variable in a register or replace it with a constant to save space.

**Interrupt Service Routine:** Can be run at any point in your main code. It interrupts the main code to do something else, which can lead to tricky trace errors if you don’t control when variables are being accessed.

**Variable Access:** Issues can happen because of this interrupt. What if a two-byte integer changes the first byte, is halfway interpreted, and then changes the second byte and the rest is interpreted? The value could be screwed up. **Int, long, and float**are multi-byte data types, so we just turn off the interrupt s during access using a couple of methods.

**Fast Interrupt Service Routines:** The microprocessor disables further interrupts while running the ISR. So we should make it as fast as possible. Therefore, we record an event but don’t attempt to process it inside the ISR, rather we do it in the main loop.

**Bit Flags:** A single-byte data type won’t be interrupted while it’s being accessed, byte’s are quantized units for the purpose of data access.

How to generate “PWM” signals to control ESCs and Servos:

**PWM:** Stands for Pulse Width Modulation is a square wave signal with a variable duty time (time of being on vs. off). We use it to simulate a **PPM** (Pulse-position modulation) signal that is used to tell ESCs their throttle state or servos their position. PWM signal’s duty time should be between 1 and 2 milliseconds.

**MCU:** microcontroller (the Arduino board) unit?

**Hardware PWM:** done by a macro of the MCU. analogWrite(pin, duty time) does this. However, it uses a large amount of library code and works only with 8-bit resolution. Hardware PWM works only on some specified pins. **Occur in response to an external event, like an input pin changing**

**Software PWM:** We use it, but only if there aren’t enough HWPWM pins available. Basically, make a loop which alternates digitalWrite with a short delay. We have to make it complicated, though, because we need the main loop to do a bunch of other stuff. So we use interrupts that interrupt at the moment they need to be processed. **Occur in response to an instruction sent in software.**

Secrets of Arduino PWM:

**Simple PWM techniques:** Using analogWrite makes it pretty easy. But there are other options, like…

**Bit-banging PWM:** The manual technique illustrated earlier

**Using Registers Directly:** (May not apply to Leonardo.)

Tutorial on Arduino Interrupts:

An interrupt is a signal that interrupts the current processor activity. Once triggered, it pauses processor activity and does a different function

**Interrupt Service Routine:** The function run when the interrupt happens. The asynchronicity provides a key advantage.

**To use an interrupt:** Set the “AVR” (microprocessor chip)’s Global Enable Interrupts bit in Status Register, set the interrupt enable bit for our specific interrupt “vector” (trigger), write an ISR (interrupt service routine.)