Lecture 11: Reading Analog Data and Intro to Interrupts

Vikram Iyer

Adapted from material by Blake Hannaford

Announcements and reminders

- Lab 2 is posted, due 5/8
  - Deadline extended to Monday
  - Tip: for scheduling, start with simple tasks

May

Monday	Tuesday	Wednesday	Thursday	Friday
01 09:00-11:00 OH (Vicente) ECE 345 13:30-15:30 OH (Kyle) ECE 345	02 09:00-11:00 OH (Kyle) ECE 345	03 09:00-11:00 OH (Vicente) ECE 345 12:30-13:50 Lecture MOR 230 *Scheduling IV* 14:00-15:00 OH (Vikram) ECE 345	04 09:00-11:00 OH (Kyle) ECE 345 13:00-15:00 OH (Kyle) ECE 345	05 12:30-13:50 Lecture MOR 230 *Intro to Lab 3* 14:00-15:00 OH (Vikram) ECE 345
08 09:00-11:00 OH (Vicente) ECE 345 23:59 Lab 2 due	09 09:00-11:00 OH (Kyle) ECE 345 13:30-15:30 OH (Joe) ECE 345	10 09:00-11:00 OH (Vicente) ECE 345 12:30-13:50 Lecture MOR 230 *Midterm Review* 14:00-15:00 OH (Vikram) ECE 345	11 09:00-11:00 OH (Kyle) ECE 345 13:30-15:30 OH (Joe) ECE 345	12 12:30-14:20 Midterm 14:00-15:00 OH (Vikram) ECE 345

Announcements and reminders

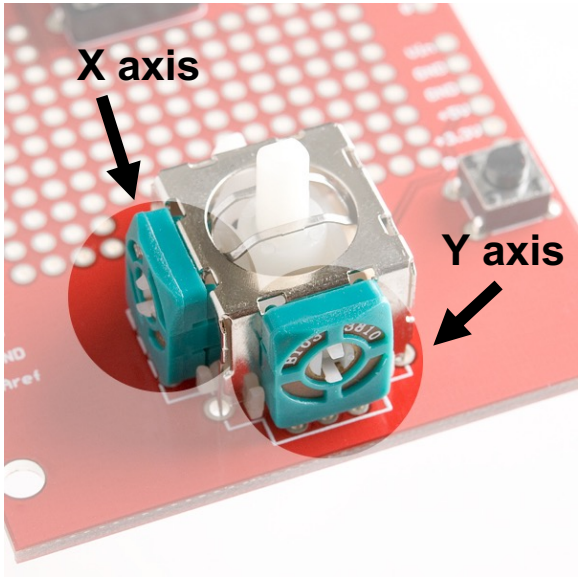
- Lab 2 is posted, due 5/8
  - Deadline extended to Monday
  - Tip: for scheduling, start with simple tasks
- Midterm on 5/12
  - Next Friday, here (MOR 230) in class
  - Everything through today
  - C programming
  - Number representation and logical operators
  - Pointers
  - Schedulers
  - Interrupts

Last time and plan for today

- Scheduling examples
- Today
  - Reading analog data and working with the joystick
  - Review context switching
  - Concept of stack
  - Introduction to interrupts

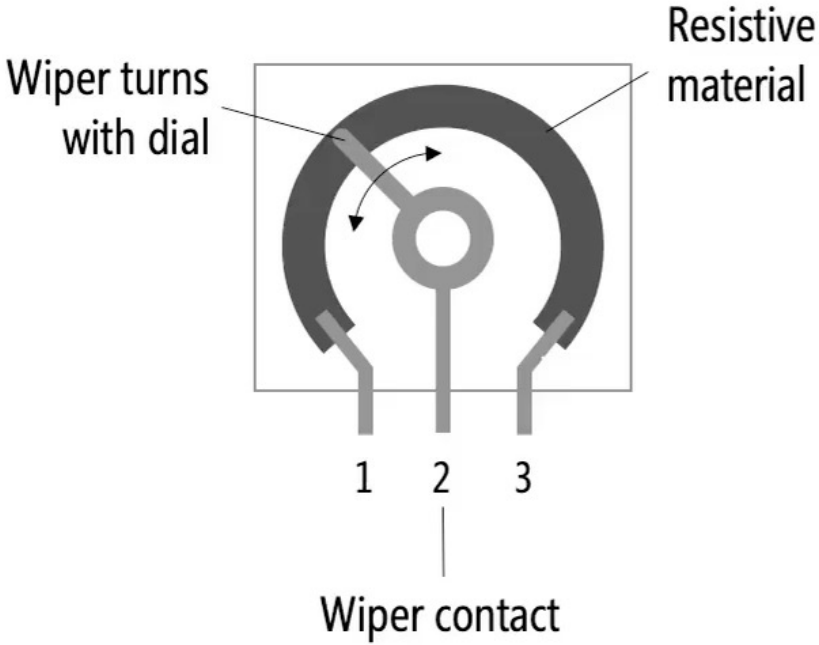
How does the thumb joystick work?

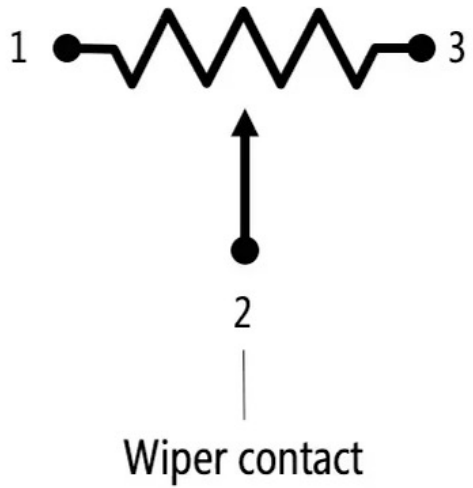
The joystick is made up of two **potentiometers** one for X and one for Y.



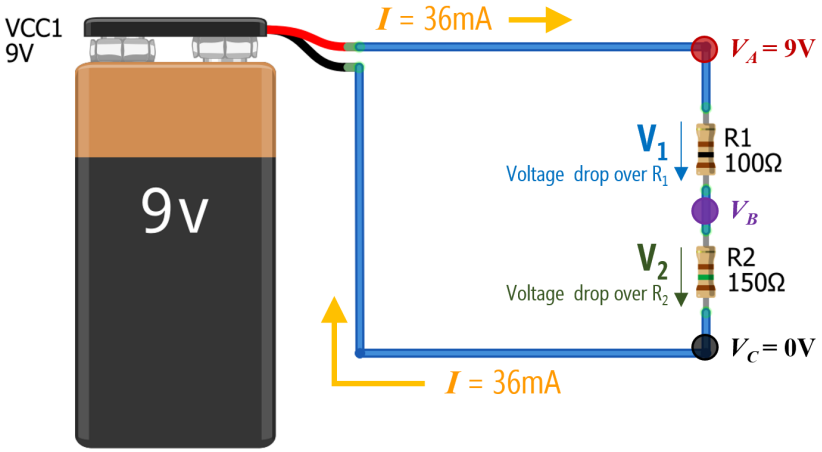
A **potentiometer** is a variable resistor

A potentiometer is a variable resistor

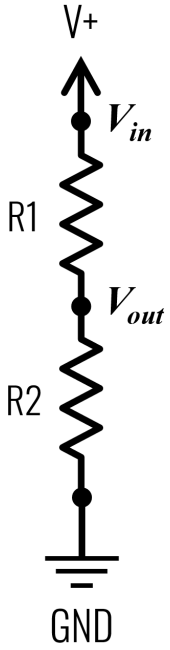




Converting changing resistance to voltage



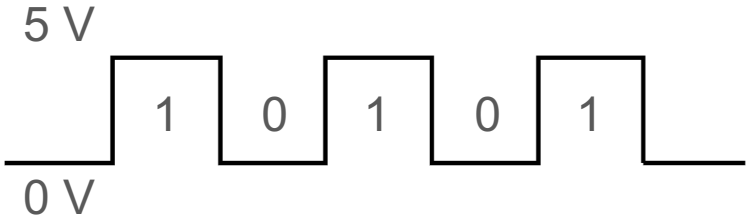
Moving the thumbstick changes the resistance = **changing voltage at V_B**



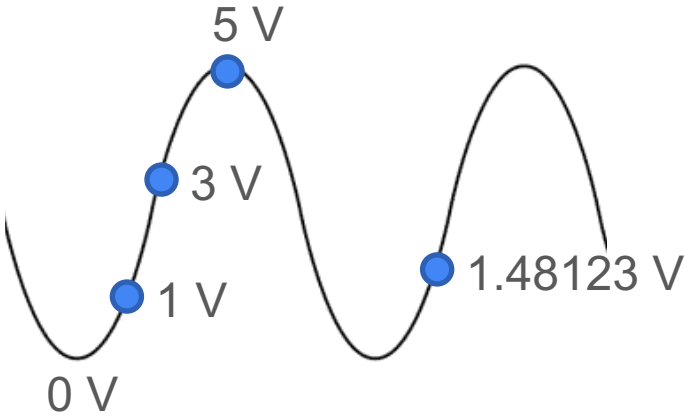
$$V_{out} = V_{in} * \frac{R_2}{R_1 + R_2}$$

Voltage divider equation

Reading voltages on the Arduino



Digital signals have only 2 voltage levels HIGH and LOW (5V, 0V) which map directly to **binary** values

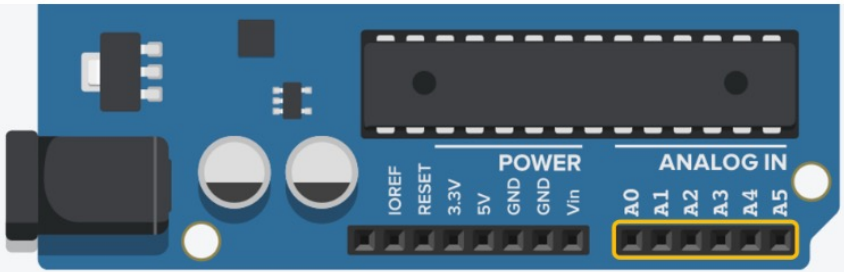


Voltage values are **continuous** or **analog**, we need to convert them into digital/binary data

Analog to digital converters (ADCs)

Specialized hardware blocks called **analog to digital converters** (ADCs) do this conversion

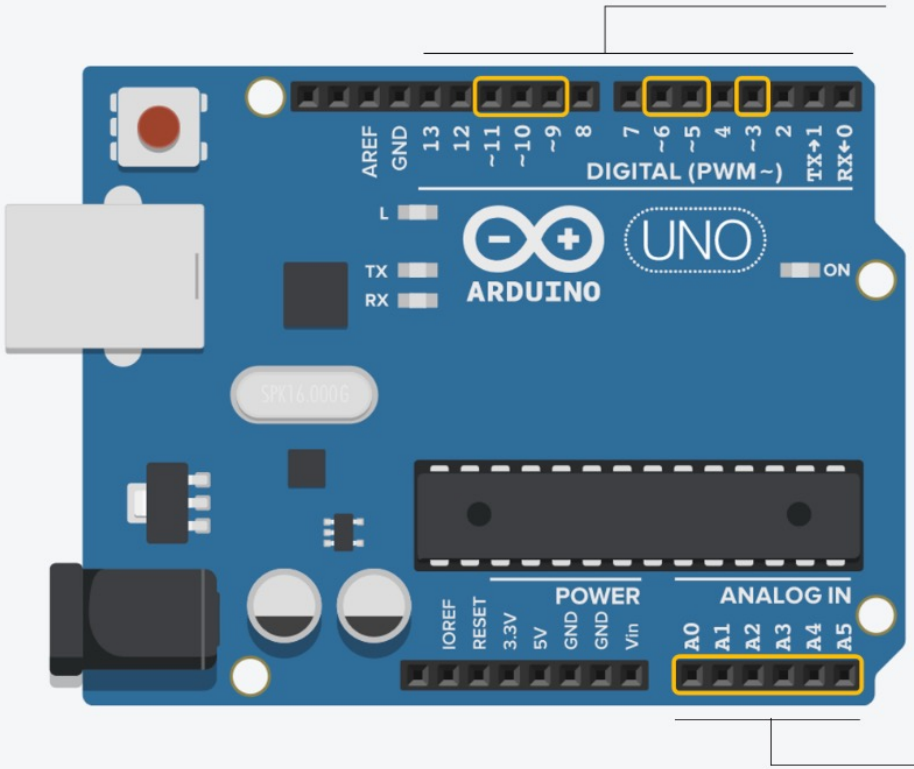
Connected to the **ANALOG IN** pins



Six Analog Input Pins

The Arduino Uno has **6** analog **input** pins, which can read a voltage signal between 0-5V using the `analogRead` command

Analog to digital converters (ADCs)



Six Analog Output Pins

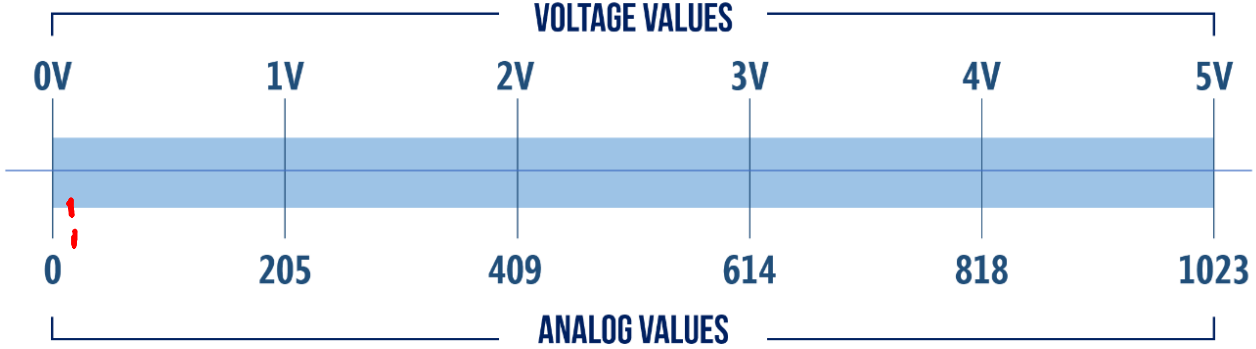
The Arduino Uno has **6** analog **output** pins indicated by the tilde ~, which can output a voltage signal between 0 and 5V using PWM via the `analogWrite` command

Beginners often confuse the **analog output** pins with the **analog input** pins. They're different!

Six Analog Input Pins

The Arduino Uno has **6** analog **input** pins, which can read a voltage signal between 0-5V using the `analogRead` command

Analog to digital converters (ADCs)




```
void loop() {
  // read input on analog pin 0:
  int sensorValue = analogRead(A0);
  // print out the value you read:
  Serial.println(sensorValue);
  delay(1); // delay in between reads
}
```

pin to sample

0-1023

The ADC maps the incoming voltage to a **10 bit** value (0 to 1023)

$\frac{5V}{1024} = 4.9\,mV \approx 5\,mV$

$2^n - 1$

sensorValue ∝ ADC counts = $\frac{V}{counts} \times$

Process Context

Process context is all the information that the process needs to keep track of its state.

Registers: Temporary storage locations for a few bytes. Typical CPU has 8-64 registers.

Program Counter: A special register which points to the next instruction to be executed.

Stack Pointer: Another special register which points to the top (bottom) of a stack of data

Processor Flags: Hardware bits in the CPU which detect errors or arithmetic results.

Context Switching

When we call a function or when an interrupt happens, we need to save the current context because we are starting a new one.

This happens a lot so it must be efficient.

- After function completes, must know where to resume.

- New process can mess up information in old process. (i.e. what if they both use the same register?)

- Saving the context and restoring it is called context switching.

Context Switching

Procedure: To switch contexts from Process 1 to Process 2:

1. Control is passed from Proc1 to "OSH"
2. OSH saves processor state from CPU registers(immediately!)
3. OSH finds stored state of Proc2 and restores it into CPU registers. IR is restored last.
4. CPU jumps to Proc2 code according to IR.

OSH = Operating System or Hardware
IR = Instruction register (current instruction that is executing)

Context Switching Types

1. **Function call**- user defined functions or to OS

2. **Interrupt**- Hardware initiated context switch

Calls

User functions (e.g. `sin(theta)`).
Main program context is saved, `sin(theta)` function is loaded.

System Calls Examples

- I/O such as write block of data to disk or apply a value to D/A converter.
- Process Control: halt, wait (pend), start, kill, or unblock another process
- Interprocess Communication: pipes, mailboxes, semaphores
- Return from Interrupt.

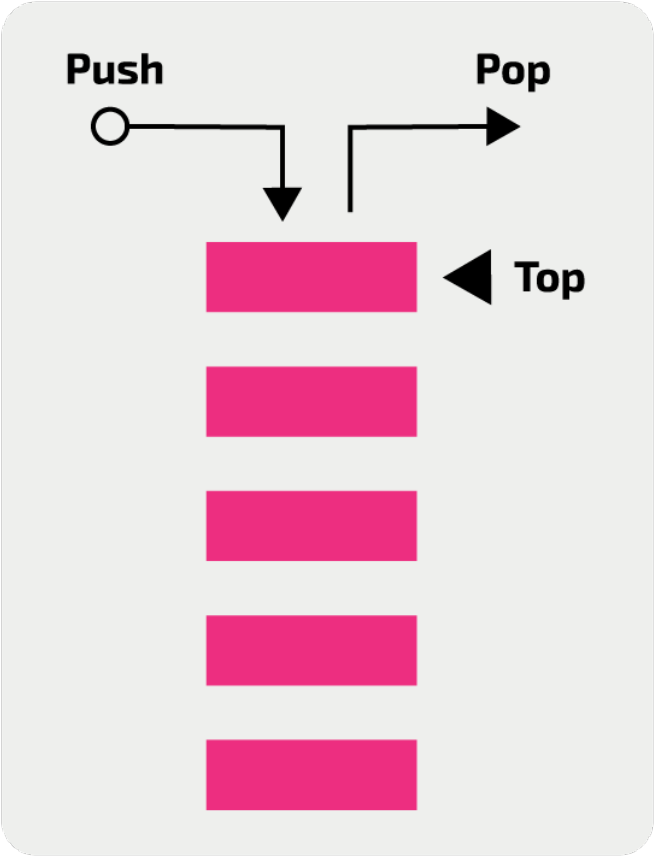
Stack

Problem: How to allocate space for context storage and keep track?

Stack

Problem: How to allocate space for context storage and keep track?

Solution: Use a data structure called a stack

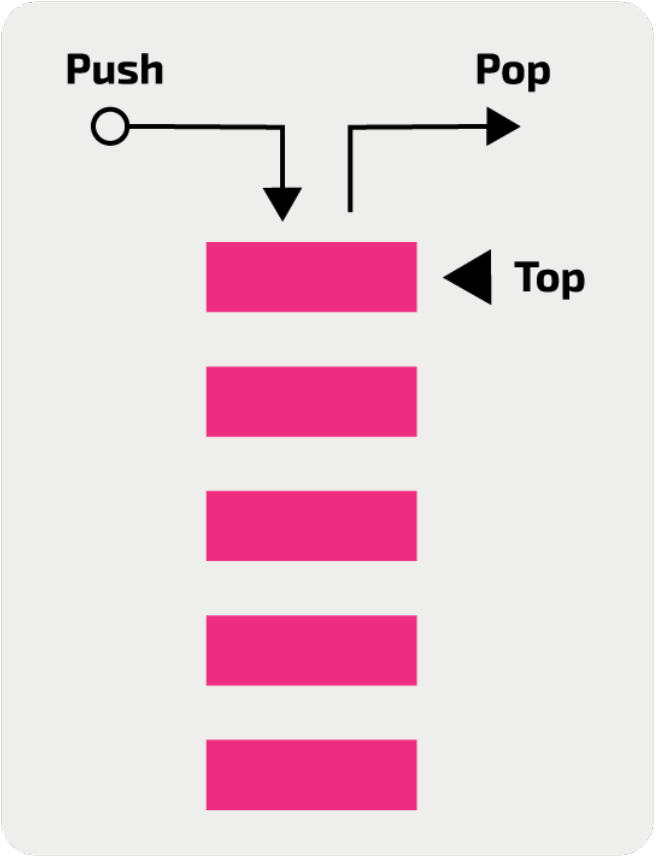


Stack

Problem: How to allocate space for context storage and keep track?

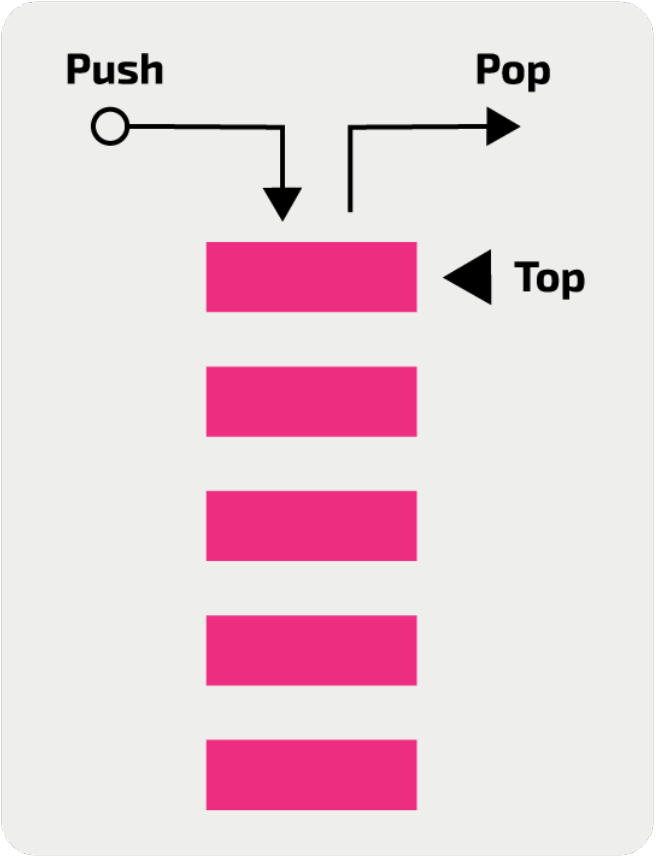
Solution: Use a data structure called a stack

- **Push**: a piece of information onto the top of the stack
- **Pop**: a piece of information off of top of the stack
- Always put/take info from current stack top indicated by the **Stack Pointer**



Stack

- **Push:** `(*SP++)` = piece of information
- **Pop:**
  `SP--;`
  piece of information = *SP
- **Initialize Stack:** SP = stack_top
- Stack grows up
- Have to allocate space for maximum stack depth



Function call

1. Push arguments to stack: `foreach i (*SP)-- = argument(i);`
2. Generate `JSR` machine instruction.
   (a) Push machine registers onto stack.
   (b) Load function addr into PC
   (c) execute next instruction (i.e. jump to *PC).
3. Function looks for arguments at `*(SP - MACH REG SIZE)`
4. When function completes, execute RET instruction.
   (a) Pop machine registers from stack and clean up args
   (b) Load old value into PC and execute next instruction

Interrupts

A context switch caused by a **hardware** event.

Types of inputs that can cause an interrupt:

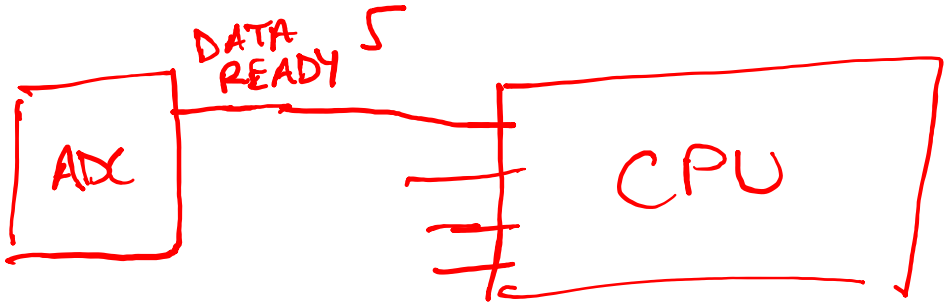
- User pushes a button or moves a mouse.
- Packet arrives on network interface.
- A sensor detects an event
- Disk drive
- A clock (`millis()` and `micros()` function use interrupts)
- A hardware timer

Interrupts

Interupts are logic signals physically wired to the CPU

Example:

- 7 lines leading into CPU.
- Peripherals can pull one line low
- A small address bus identifies the specific interrupt within a line.
- Lines are ranked by priority



Interrupts

Interrupt Vectors: Address in memory the computer goes to. Each interupt has a specific code number which is applied to the processor interrupt address bits (maybe 8bits)

CPU has a specific hard-wired memory address called the **interrupt vector table**. The interrupt vector table is an array of function pointers.

The functions they point to are called **interrupt service routines, ISRs.** Each hardware device is hooked up to a specific interrupt vector table address. We need to write each ISR to handle the proper device.

Table 14-1. Reset and Interrupt Vectors

Vector No.	Program Address[2]	Source	Interrupt Definition
1	$0000[1]	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$0002	INT0	External Interrupt Request 0
3	$0004	INT1	External Interrupt Request 1
4	$0006	INT2	External Interrupt Request 2
5	$0008	INT3	External Interrupt Request 3
6	$000A	INT4	External Interrupt Request 4
7	$000C	INT5	External Interrupt Request 5
8	$000E	INT6	External Interrupt Request 6
9	$0010	INT7	External Interrupt Request 7
10	$0012	PCINT0	Pin Change Interrupt Request 0
11	$0014	PCINT1	Pin Change Interrupt Request 1
12	$0016[3]	PCINT2	Pin Change Interrupt Request 2
13	$0018	WDT	Watchdog Time-out Interrupt
14	$001A	TIMER2 COMPA	Timer/Counter2 Compare Match A
15	$001C	TIMER2 COMPB	Timer/Counter2 Compare Match B
16	$001E	TIMER2 OVF	Timer/Counter2 Overflow

Setting up an interrupt

Hardware:

- For external interrupts connect wires to the right pins

Software:

- Write an ISR
- Copy the pointer in to the interrupt vector table

In practice we'll use Arduino's ISR macro which will copy it in and also apply some code before and after to save and reload state to make sure you don't mess up other built in Arduino functions. Other chips have similar functions or macros

Sample Code: Pin change interrupt

```
ISR (PCINT0_vect) {
  // handle pin change interrupt for D8 to D13 here
} // end of PCINT0_vect

void setup () {
  // pin change interrupt (example for D9)
  PCMSK0 |= bit (PCINT1); // want pin 9
  PCIFR |= bit (PCIF0); // clear any outstanding interrupts
  // enable pin change interrupts for D8 to D13
  PCICR |= bit (PCIE0);
}

void loop(){ // we can have programs with an empty/minimal loop
}
```

Interrupt service routine

setting register bits

Servicing an interrupt

Hardware Level- Upon hardware interrupt…

1. Push processor context onto stack.
2. Load appropriate address from IVT into program counter (PC).
3. jump to *PC.

Interrupt vector table

Software Level

1. (There are no arguments)
2. Block (“mask”) all other interrupts if necessary
3. “Service” the device (run ISR function’s code)
4. Unmask other interrupts
5. Execute a return from the interrupt

HW pops processor context and continues the program

ISR Programming

Problems with ISRs

- ISRs are virtually impossible to debug! Many debugging tools will either break the code or the interrupt will break the debugger
- ISRs take over the processor – even pre-emptive schedulers
- ISRs can easily mess up other processes if they write in wrong global variables
- take up a lot of CPU time
- play around with the stack

ISR Programming

Tips to avoid these problems

- Keep your ISRs small, simple, & quick
- No loops inside ISR's
- No complex logic (no nested if's)
- Don't do unrelated I/O in the ISR (for example, no printf)
- Guidelines: Max: 1/2 page of C, 1 page of ASM
- Do not trust / use debugger.
- Do absolute minimum in ISR, do the rest in a regular task.

serial.print

ISR Programming (Link)

35

TL;DR :

When writing an Interrupt Service Routine (ISR):

- Keep it *short*
- Don't use `delay ()`
- Don't do serial prints
- Make variables shared with the main code **volatile**
- Variables shared with main code may need to be protected by "critical sections" (see below)
- Don't try to turn interrupts off or on

https://arduino.stackexchange.com/questions/30968/how-do-interrupts-work-on-the-arduino-uno-and-similar-boards

Volatile variables

A variable should only be marked **volatile** if it is used both inside an ISR, and outside one.

- Variables **only** used outside an ISR should **not** be volatile.
- Variables **only** used inside an ISR should **not** be volatile.
- Variables used both inside and outside an ISR **should** be volatile.

Marking a variable as volatile tells the compiler to not "cache" the variables contents into a processor register, but always read it from memory, when needed. This may slow down processing, which is why you don't just make every variable volatile, when not needed.

Critical sections

```
volatile unsigned int count;

ISR (TIMER1_OVF_vect){ count++; } // end TIMER1_OVF_vect

void setup(){ pinMode (13, OUTPUT); } // end of setup

void loop () {
    noInterrupts (); // <------ critical section
    if (count > 20)
        digitalWrite (13, HIGH);
    interrupts (); // <------ end critical section
} // end of loop
```

set register to disable interrupts

count == 20