MRSD Systems Engineering and Management for Robotics: Introduction to Microcontrollers

John M. Dolan

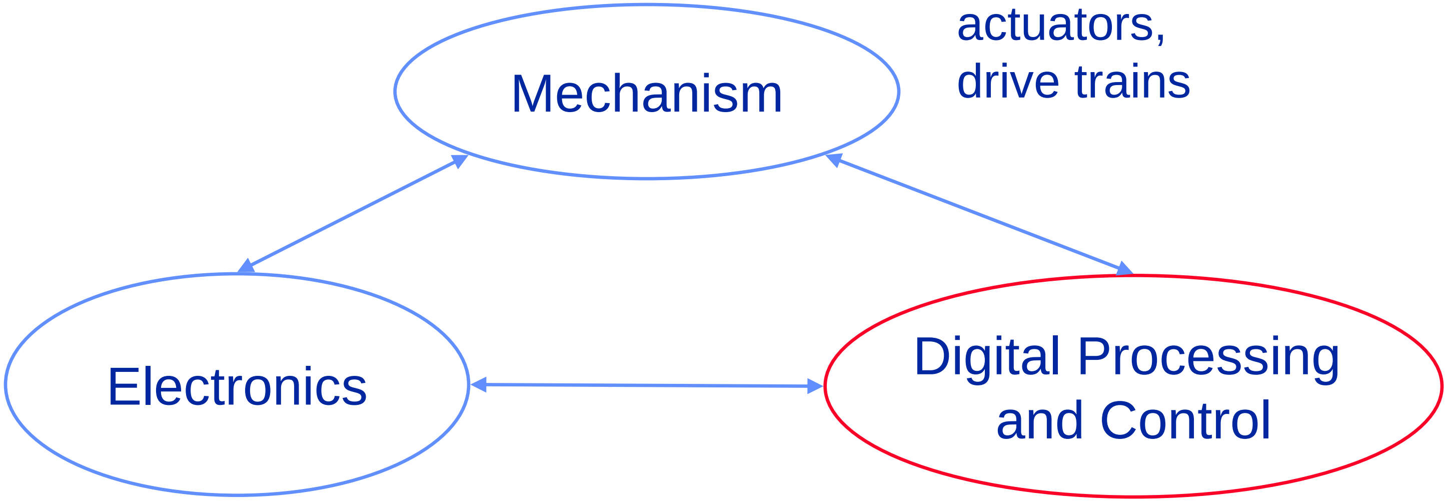
Outline

- Microcontroller Overview
- Event-Based Programming
  - State machines
  - Polling & Interrupts
- Microcontroller Digital I/O
  - On/off I/O
    - LEDs, solenoids, switches
  - Timers
  - Pulse-based I/O
    - PWM motor driving
    - Encoder position feedback
    - Serial and interprocessor communications

Programming/Control Aspect

Mechatronics is the synergistic integration of mechanism, electronics, and computer control to achieve a functional system

Robotics…overlaps with electronics, computer science, AI, mechatronics, …



Mechanism

Electronics

Digital Processing and Control

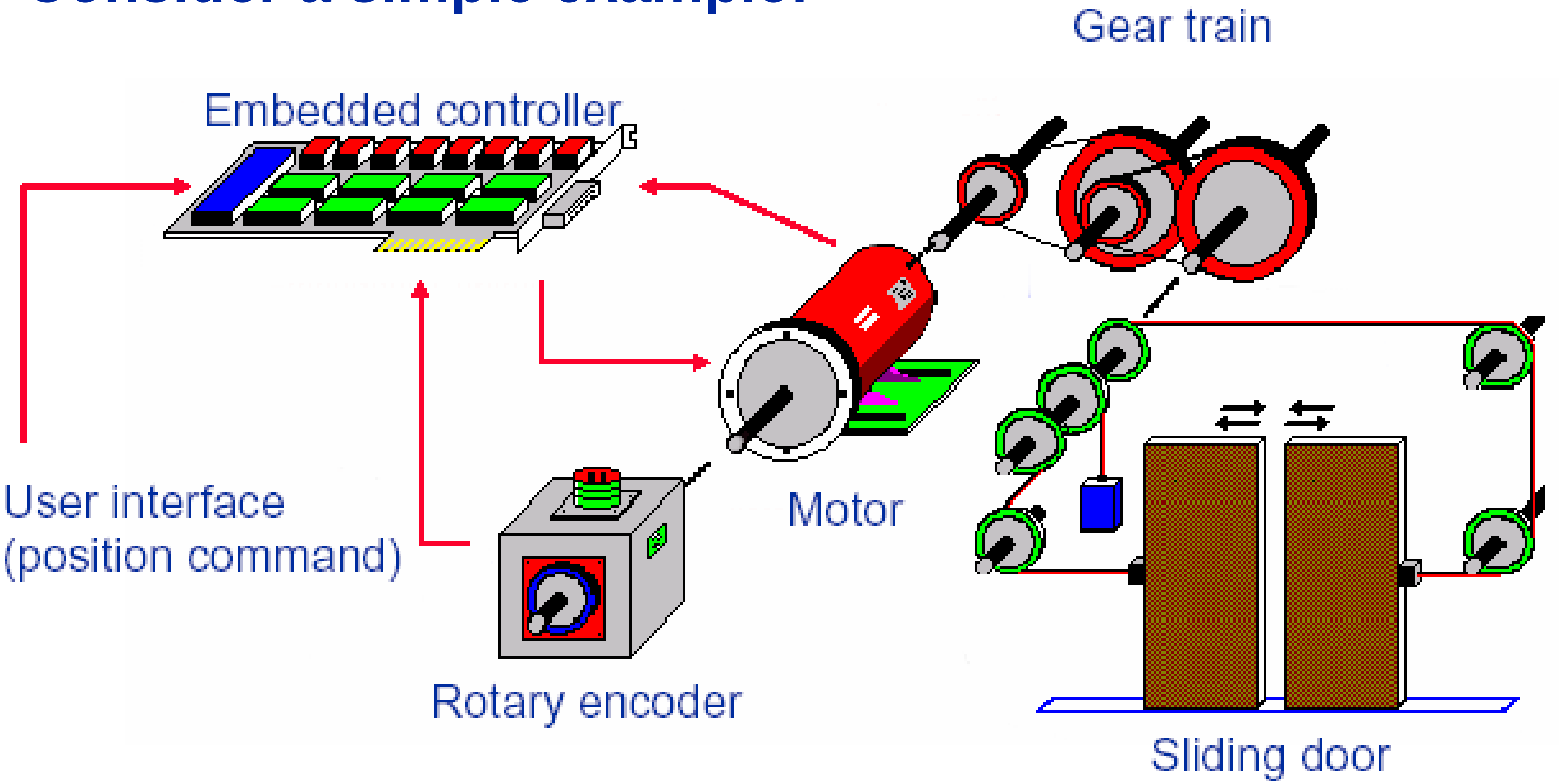
Structure,
actuators,
drive trains

Sensors,
communications,
interfacing,
power

Digital computer:
embedded controller

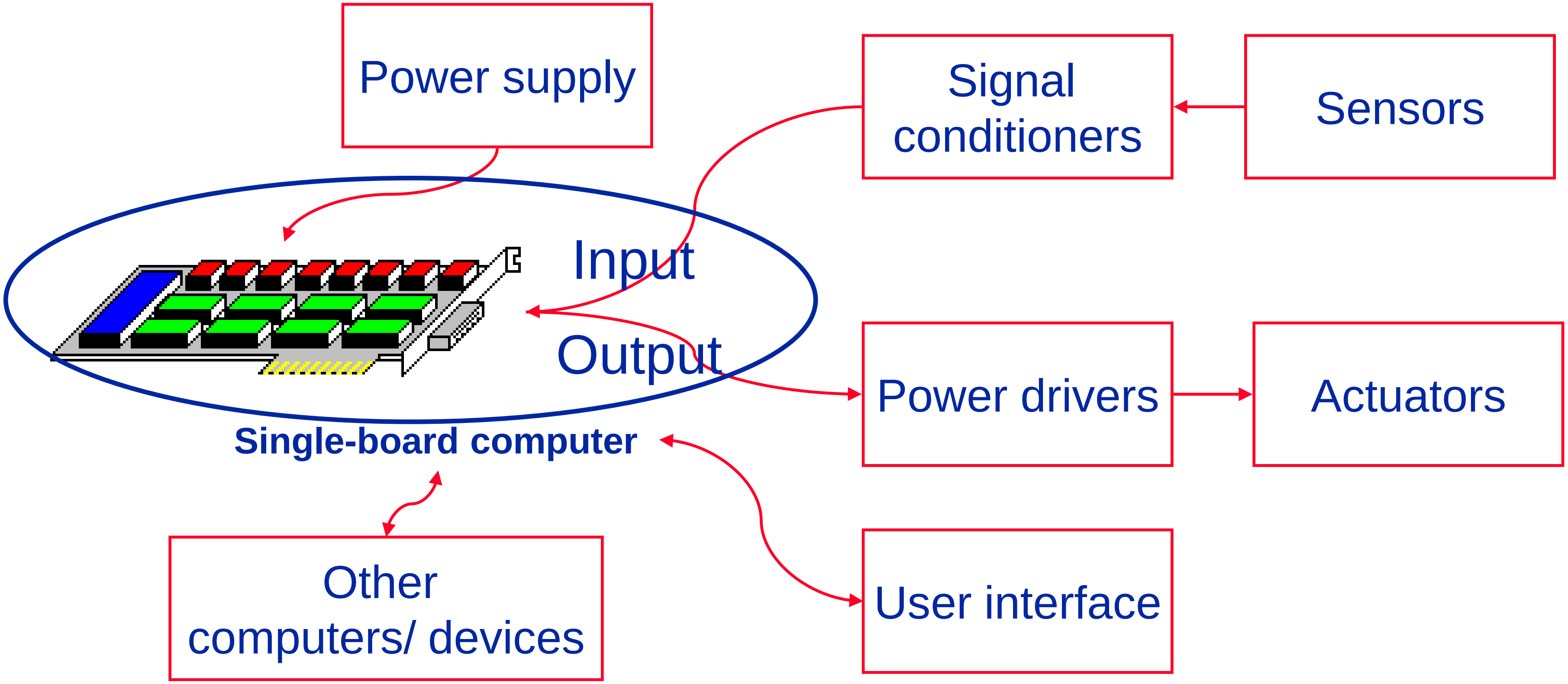
Why should we use a programmable controller?

- Consider a simple example:



- Yes, it can be done without a computer, but think about
  - additional features, increased capabilities, reconfigurability, communication, data collection…

Embedded Controllers



- **Choice of microprocessor or microcontroller**
  - **Driven by cost, power, reliability in application**
  - **Size becoming less of a factor in choice**

Microcontrollers vs. Microprocessors and DSPs

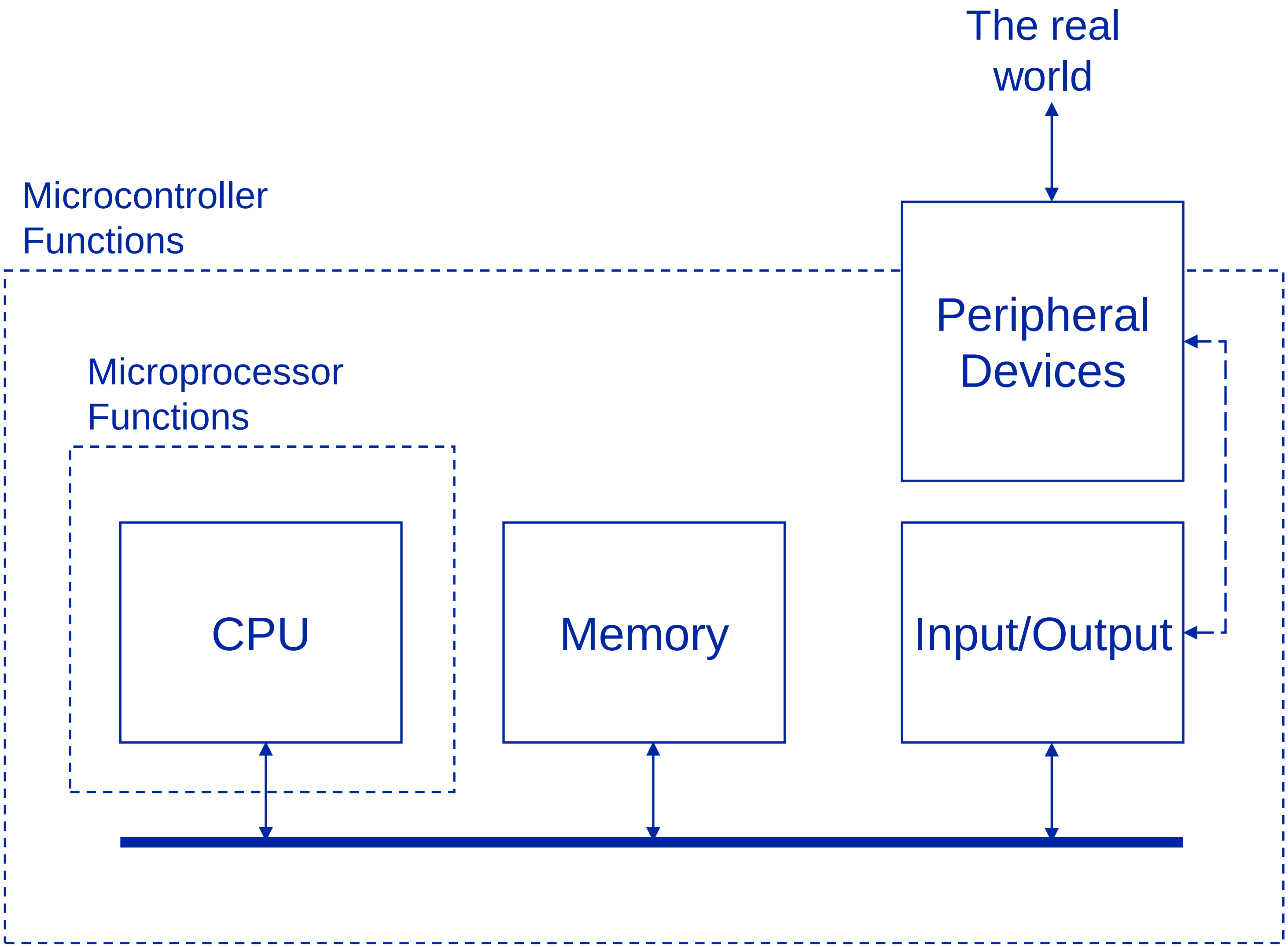
- Microprocessors
  - High-speed information processing
  - High-speed standard digital I/O and communication
  - Large memory space
  - Flexible architecture (e.g. DMA, ATA/IDE, USB, etc.)
- Microcontrollers
  - General-purpose parallel and serial I/O
  - Special functions (ADC, timers, drivers)
  - High-speed flexible interrupts
  - Small amount of on-chip RAM, ROM
  - Low-power
  - Cheap!

  Usually this is a single-chip solution!
- DSP – Digital signal processors
  - Fast recursive signal processing
  - Fast multiply and accumulate (MAC)
  - Some include floating point units

Other Devices (also "micros")

- Field-Programmable Gate Array (FPGA)
  - Large number of fundamental logical building blocks
  - Customizable
  - May be configured as microprocessor, microcontroller, DSP, etc.
- Programmable System-on-Chip (PSoC)
  - Incorporates a microcontroller
  - Arrays of *some* configurable analog and digital logic elements
- Application-Specific Integrated Circuit (ASIC)
  - Highly flexible, can incorporate whatever circuitry is needed
  - Basically an FPGA programmed at the factory

Typical bus-oriented microcomputer



Microcontroller Types

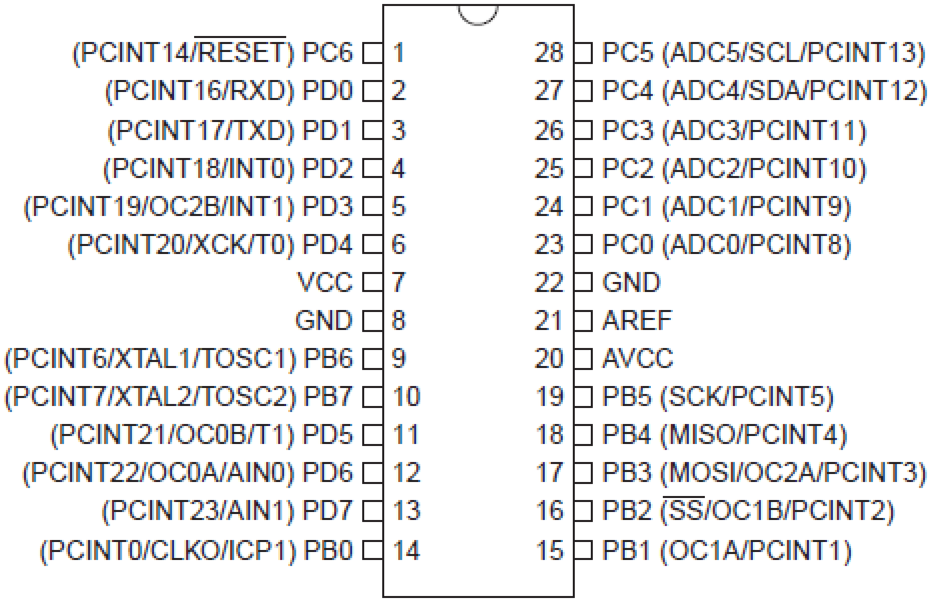
- Central Processing Unit (CPU) has Arithmetic Logic Unit (ALU) whose size strongly influences microcontroller speed and cost
- 4-bit (ALU register size) microcontrollers (e.g. Intel 4004, Intel 8008)
  - Ultra-low end of the microcontroller spectrum
  - Widely used in the 1970s
- 8-bit microcontrollers (e.g. PIC16, PIC18, $\overbrace{\text{ATmega168, ATmega328)}}^{\text{Arduino}}$
  - Low end of the modern microcontroller spectrum
  - Combine low cost (<$1) w/ reasonable capabilities
- 16-bit microcontrollers (e.g. PIC24)
  - Middle range of the microcontroller spectrum
  - Reasonable cost (~$1-$15) w/ strong capabilities
- 32-bit microcontrollers (e.g. ARM Cortex M3 [NI Luminary])
  - High end of the microcontroller spectrum
  - Competitive cost (~$1-$20) w/ superior capabilities
  - Dominated by ARM (Advanced RISC Machine) architecture

Microcontroller Subsystems & Peripherals

- Control Registers
  - Used for configuration (e.g. clock speed, Baud rate, pin usage)
- Ports and Parallel I/O
  - Groups of digital I/O; logically grouped for ease of configuration
- Counters
  - Registers that count events (e.g. encoder edge transitions)
- Timers
  - Counters that increment based on intervals of time from a clock source
- Serial I/O
  - One-bit-at-a-time inter-device communication (synch. or asynch.)
  - Various conventions: USB, CAN, SPI, I^{2}C
- Analog-to-Digital Converters
- Digital-to-Analog Converters (PWM usually used instead)

ATmega328: a single-chip solution

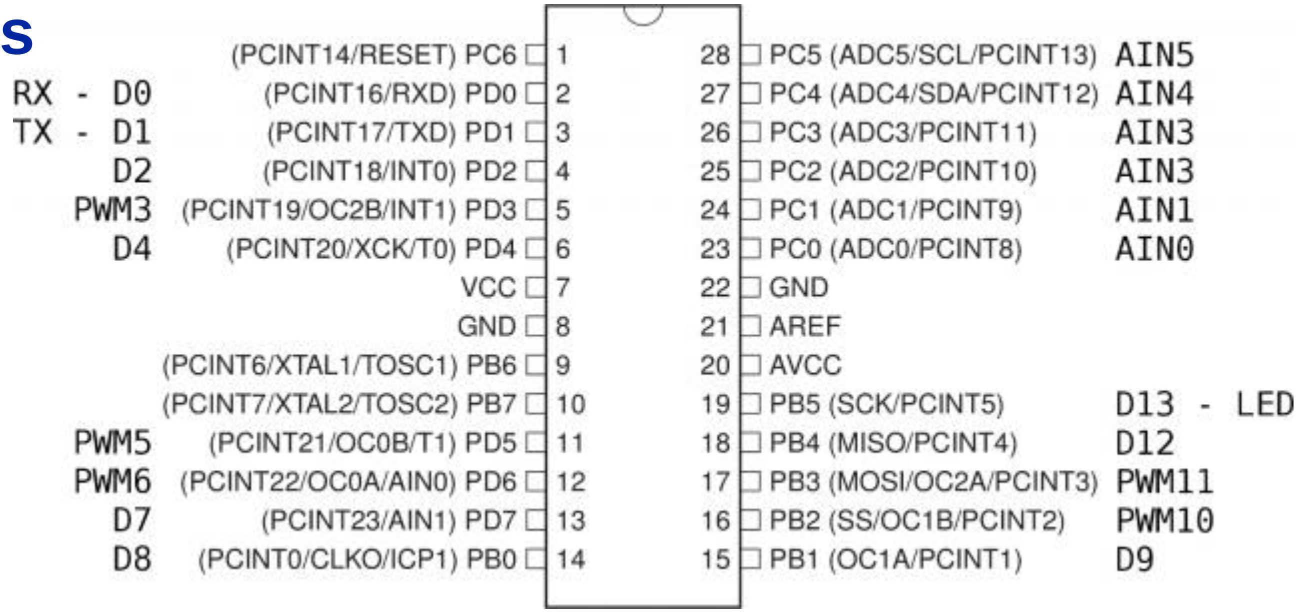
- 28-pin DIP
- Most port pins are multiplexed with alternate functional options
  - Timing inputs and outputs
  - Output Compare (OC), Input Capture (IC)
  - Pin Change Interrupt inputs (PCINT)



ATmega328: a single-chip solution

- 28-pin DIP
- Arduino Uno pinout maps specific functions to specific pins
  - Serial comms
  - Digital I/O
  - PWM
  - Analog inputs
  - LED

Arduino Uno pinout



	(PCINT14/RESET) PC6	1	28	PC5 (ADC5/SCL/PCINT13)	AIN5
RX - D0	(PCINT16/RXD) PD0	2	27	PC4 (ADC4/SDA/PCINT12)	AIN4
TX - D1	(PCINT17/TXD) PD1	3	26	PC3 (ADC3/PCINT11)	AIN3
D2	(PCINT18/INT0) PD2	4	25	PC2 (ADC2/PCINT10)	AIN3
PWM3	(PCINT19/OC2B/INT1) PD3	5	24	PC1 (ADC1/PCINT9)	AIN1
D4	(PCINT20/XCK/T0) PD4	6	23	PC0 (ADC0/PCINT8)	AIN0
	VCC	7	22	GND	
	GND	8	21	AREF	
	(PCINT6/XTAL1/TOSC1) PB6	9	20	AVCC	
	(PCINT7/XTAL2/TOSC2) PB7	10	19	PB5 (SCK/PCINT5)	D13 - LED
PWM5	(PCINT21/OC0B/T1) PD5	11	18	PB4 (MISO/PCINT4)	D12
PWM6	(PCINT22/OC0A/AIN0) PD6	12	17	PB3 (MOSI/OC2A/PCINT3)	PWM11
D7	(PCINT23/AIN1) PD7	13	16	PB2 (SS/OC1B/PCINT2)	PWM10
D8	(PCINT0/CLKO/ICP1) PB0	14	15	PB1 (OC1A/PCINT1)	D9

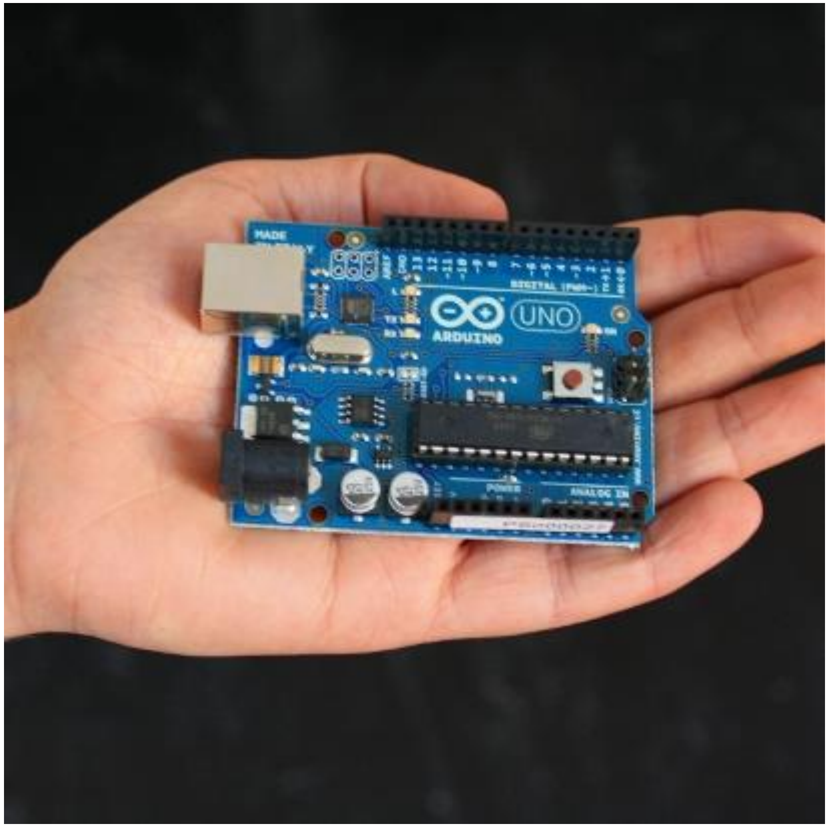
ATmega328: Features

- **ATmega328 microcontroller features**
  - **16 MHz**
  - **Program memory: 32KB FLASH**
  - **Data memory: 2KB internal SRAM; 1KB EEPROM**
  - **10-bit multiplexed analog input module (6 channels)**
  - **6 PWM channels**
  - **2 x 8-bit, 1 x 16-bit timers**
  - **23 programmable I/O lines (2 8-bit ports, 1 7-bit)**
  - **32 general-purpose registers**

ATmega328: Programming

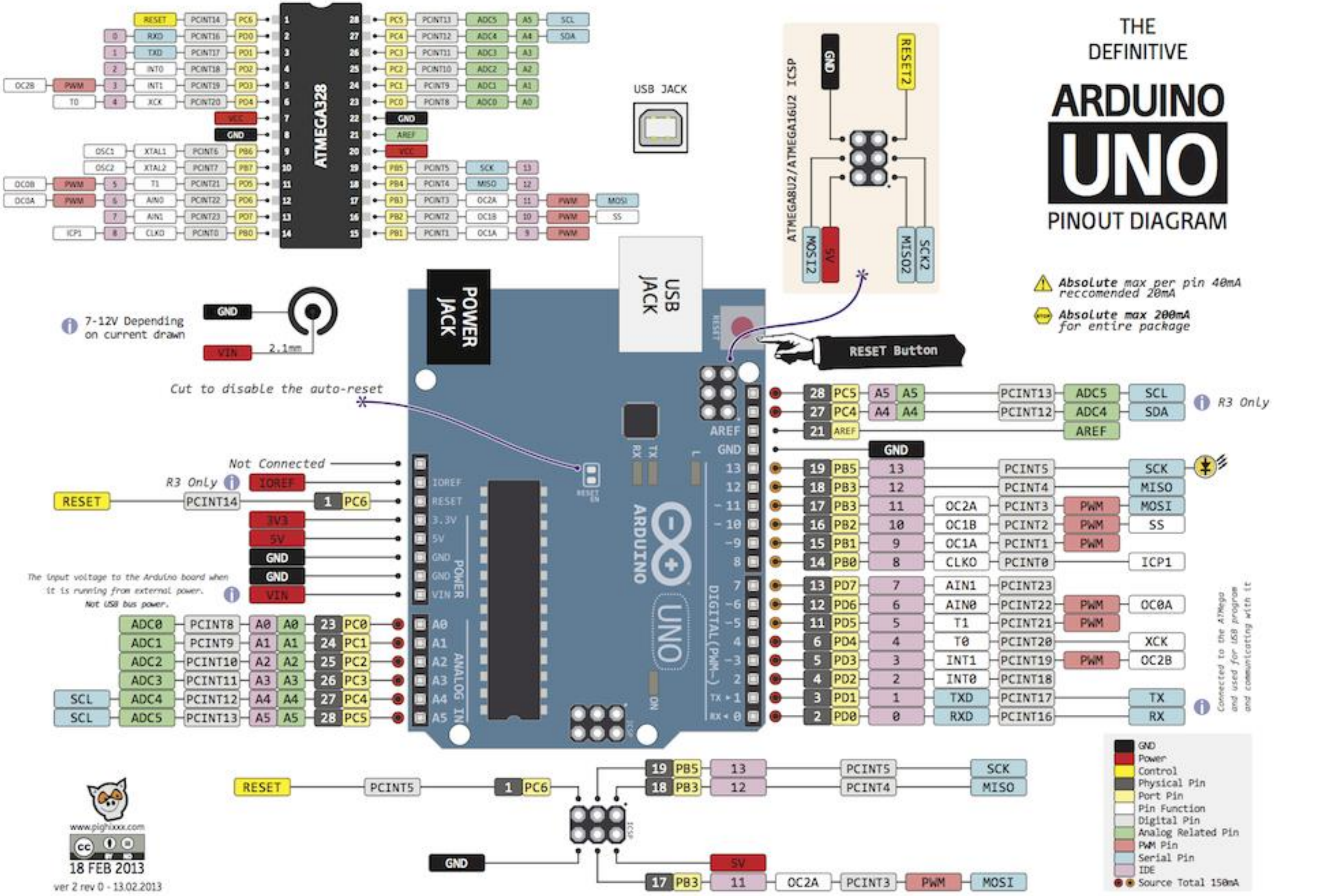
- RISC CPU
- 131 instructions to learn
  - Push Around Bits/Bytes
  - Comparison, Branching, Subroutine
  - Bitwise Logical Operations
  - Add/Subtract 8-bit Integers
  - Everything else must be EMULATED
    - Anything with floating point data
    - Transcendentals (sin, cos, exponents) are painfully slow
- *C compiler available as freeware*
  - ***No need to use assembly language, but you are certainly welcome to if you want.***

Development: The Arduino



Arduino Uno: hardware board built around an 8-bit AVR microcontroller

Arduino Uno: Built on the ATmega328

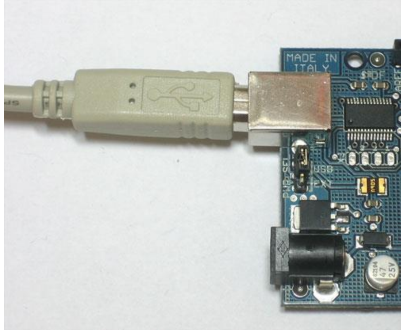


THE DEFINITIVE
ARDUINO
UNO
PINOUT DIAGRAM
ATMEGA328
USB JACK
POWER JACK
ATMEGA8U2/ATMEGA16U2 ICSP
GND
RESET2
MOSI2
5V
MISO2
SCK2
Absolute max per pin 40mA reccomended 20mA
Absolute max 200mA for entire package
7-12V Depending on current drawn
GND
VIN
2.1mm
RESET Button
Cut to disable the auto-reset
Not Connected
R3 Only
IOREF
RESET
PCINT14
1 PC6
3V3
5V
GND
GND
VIN
The input voltage to the Arduino board when it is running from external power. Not USB bus power.
ADC0 PCINT8 A0 A0 23 PC0
ADC1 PCINT9 A1 A1 24 PC1
ADC2 PCINT10 A2 A2 25 PC2
ADC3 PCINT11 A3 A3 26 PC3
SCL ADC4 PCINT12 A4 A4 27 PC4
SCL ADC5 PCINT13 A5 A5 28 PC5
28 PC5 A5 A5 PCINT13 ADC5 SCL
27 PC4 A4 A4 PCINT12 ADC4 SDA
R3 Only
21 AREF AREF
GND
19 PB5 13 PCINT5 SCK
18 PB3 12 PCINT4 MISO
17 PB3 11 OC2A PCINT3 PWM MOSI
16 PB2 10 OC1B PCINT2 PWM SS
15 PB1 9 OC1A PCINT1 PWM
14 PB0 8 CLKO PCINT0 ICP1
13 PD7 7 AIN1 PCINT23
12 PD6 6 AIN0 PCINT22 PWM OC0A
11 PD5 5 T1 PCINT21 PWM
6 PD4 4 T0 PCINT20 XCK
5 PD3 3 INT1 PCINT19 PWM OC2B
4 PD2 2 INT0 PCINT18
3 PD1 1 TXD PCINT17 TX
2 PD0 0 RXD PCINT16 RX
Connected to the ATmega and used for USB program and communicating with it
RESET PCINT5 1 PC6
19 PB5 13 PCINT5 SCK
18 PB3 12 PCINT4 MISO
GND
5V
17 PB3 11 OC2A PCINT3 PWM MOSI
ICSP
GND
Power
Control
Physical Pin
Port Pin
Pin Function
Digital Pin
Analog Related Pin
PWM Pin
Serial Pin
IDE
Source Total 150mA
www.pighixxx.com
18 FEB 2013
ver 2 rev 0 - 13.02.2013

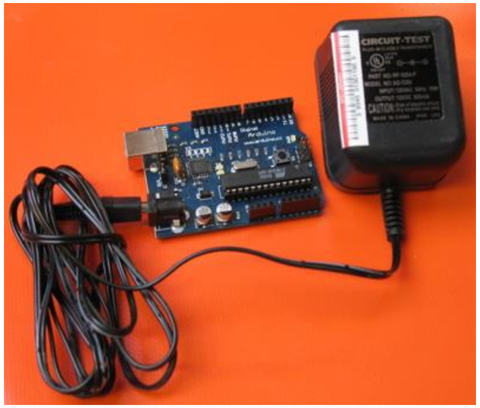
Arduino: Electrical Characteristics

- Power
  - USB (5V) connection OR “external” power supply
    - External: 7-12V (>= 250 mA) via barrel connector OR VIN pin
  - Microcontroller has a sleep mode
- Digital I/O current limit
  - High sink/source current: 40 mA per pin, but…
    - Note: off USB, don’t use any high-power devices (DC/Stepper motors, solenoids, etc.)
    - Use a separate supply for high-power devices!
  - Do not short output pins!

Power via USB



Power via DC adapter



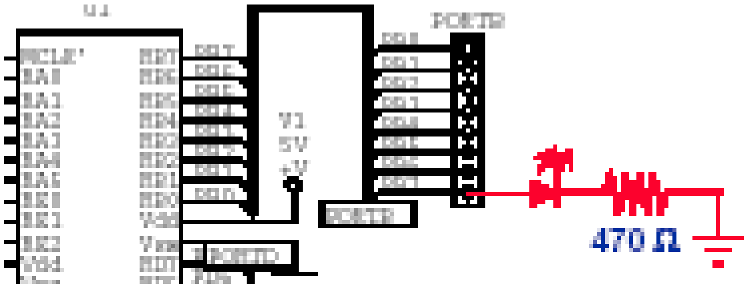
What Does a Program Look Like?

"Include" and other pre-compiler statements (none needed here)

```
void setup() {                    // Runs once
  // initialize the digital pin as an output
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(13, OUTPUT);
}
```

```
void loop() {                      // Repeats
  digitalWrite(13, HIGH);          // set the LED on
  delay(1000);                     // wait for a second
  digitalWrite(13, LOW);           // set the LED off
  delay(1000);                     // wait for a second
} // http://arduino.cc/en/Tutorial/Blink
```

main program body

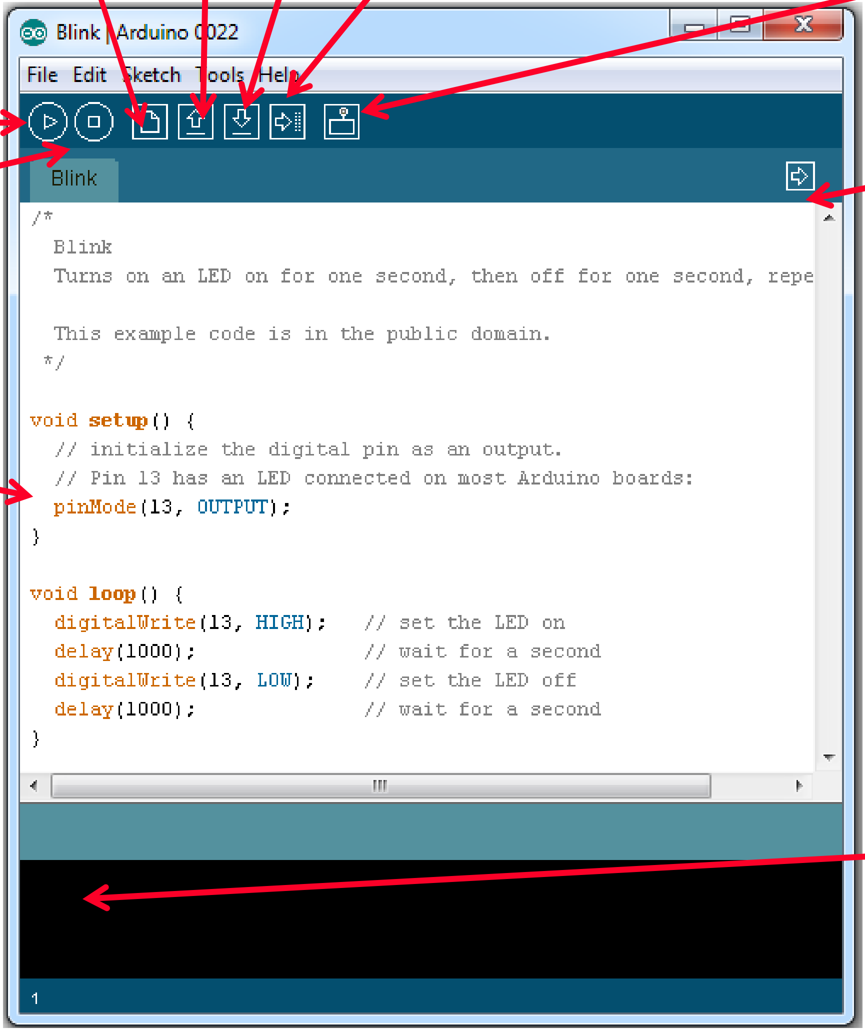


software is closely related to hardware

Arduino Development Environment

- Integrated Development Environment (IDE)
    - Write, compile, build, and transfer programs
        - Create new programs
        - Test built-in example code
        - Find new examples online
            - If you can think of it, it has probably been done before. If you use somebody else’s code, CITE IT

Using the Arduino IDE



```
/*
  Blink
  Turns on an LED on for one second, then off for one second, repe

  This example code is in the public domain.
 */

void setup() {
  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);   // set the LED on
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // set the LED off
  delay(1000);              // wait for a second
}
```

Verify (aka compile)

Stop

New

Open

Save

Upload to board (also compiles/builds)

Serial monitor

Tab Options

http://liudr.wordpress.com/2011/02/16/using-tabs-in-arduino-ide/

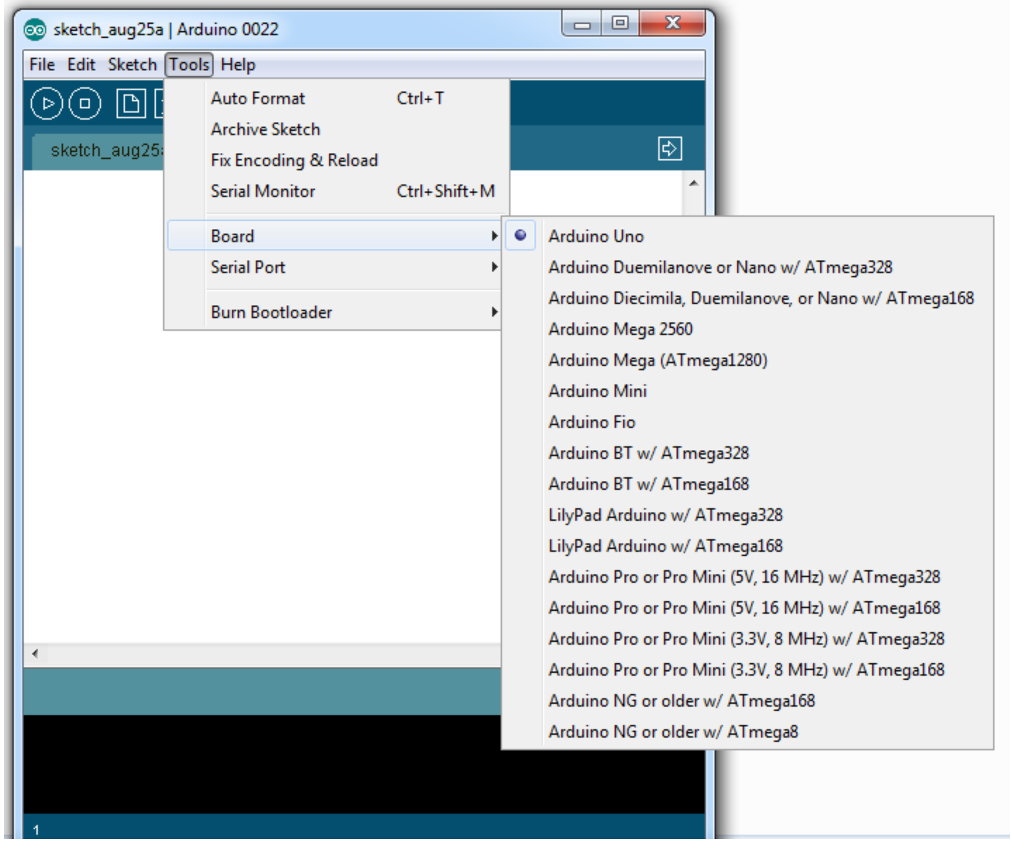
Edit source code here

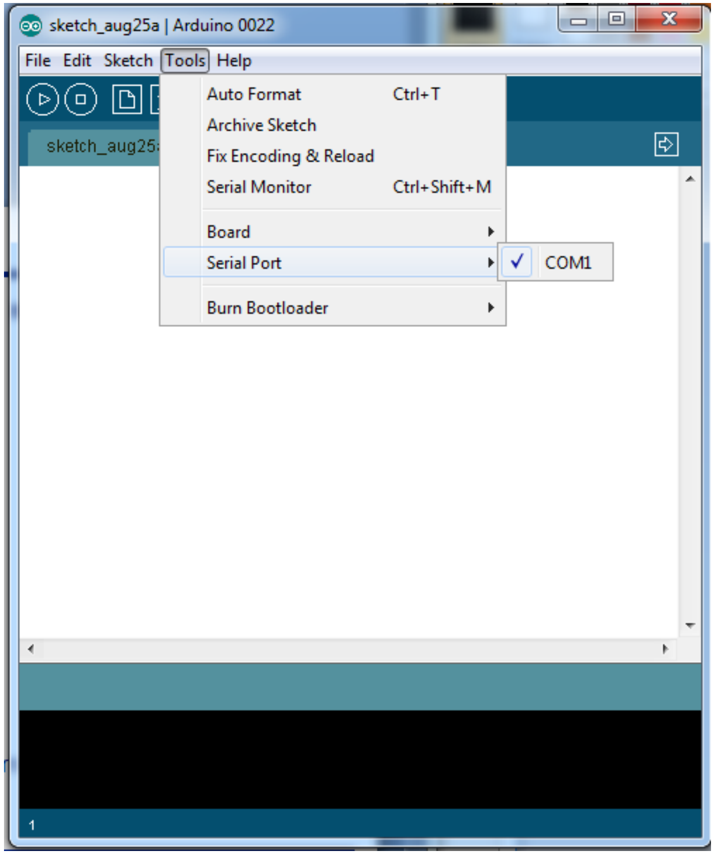
Download from: http://arduino.cc/en/main/software

Serial monitor

Arduino – Setting Up

- Ensure the right type of Arduino is selected, and the right serial port


sketch_aug25a | Arduino 0022
File Edit Sketch Tools Help
sketch_aug25a
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Serial Monitor Ctrl+Shift+M
Board
Serial Port
Burn Bootloader
● Arduino Uno
Arduino Duemilanove or Nano w/ ATmega328
Arduino Diecimila, Duemilanove, or Nano w/ ATmega168
Arduino Mega 2560
Arduino Mega (ATmega1280)
Arduino Mini
Arduino Fio
Arduino BT w/ ATmega328
Arduino BT w/ ATmega168
LilyPad Arduino w/ ATmega328
LilyPad Arduino w/ ATmega168
Arduino Pro or Pro Mini (5V, 16 MHz) w/ ATmega328
Arduino Pro or Pro Mini (5V, 16 MHz) w/ ATmega168
Arduino Pro or Pro Mini (3.3V, 8 MHz) w/ ATmega328
Arduino Pro or Pro Mini (3.3V, 8 MHz) w/ ATmega168
Arduino NG or older w/ ATmega168
Arduino NG or older w/ ATmega8
1



You can find out what serial port the Arduino is on by plugging and unplugging the Arduino, and seeing how the Tools→Serial Port list changes.

Outline

- Microcontroller Overview
- Event-Based Programming
    - State machines
    - Polling & Interrupts
- Microcontroller Digital I/O
    - On/off I/O
        - LEDs, solenoids, switches
    - Timers
    - Pulse-based I/O
        - PWM motor driving
        - Encoder position feedback
        - Serial and interprocessor communications

State machines

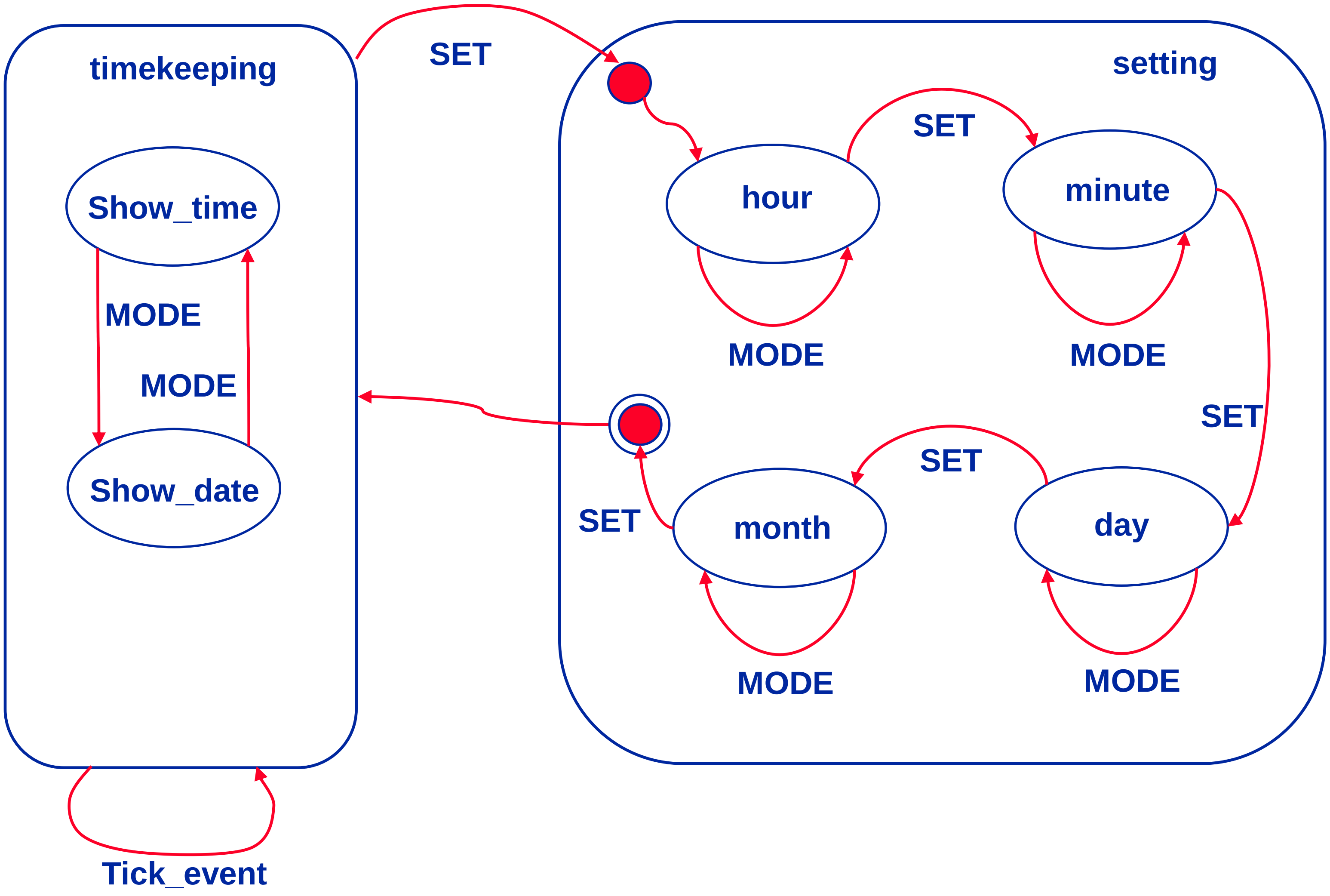
- Useful abstraction for systems with discrete states
  - States are engaged based on events sensed in the world
- To set up a state machine:
  - Define the events
  - Define the states
  - Determine order of transition between states
  - Set the initial state of the state machine
- Example: let us consider a wristwatch

MODE

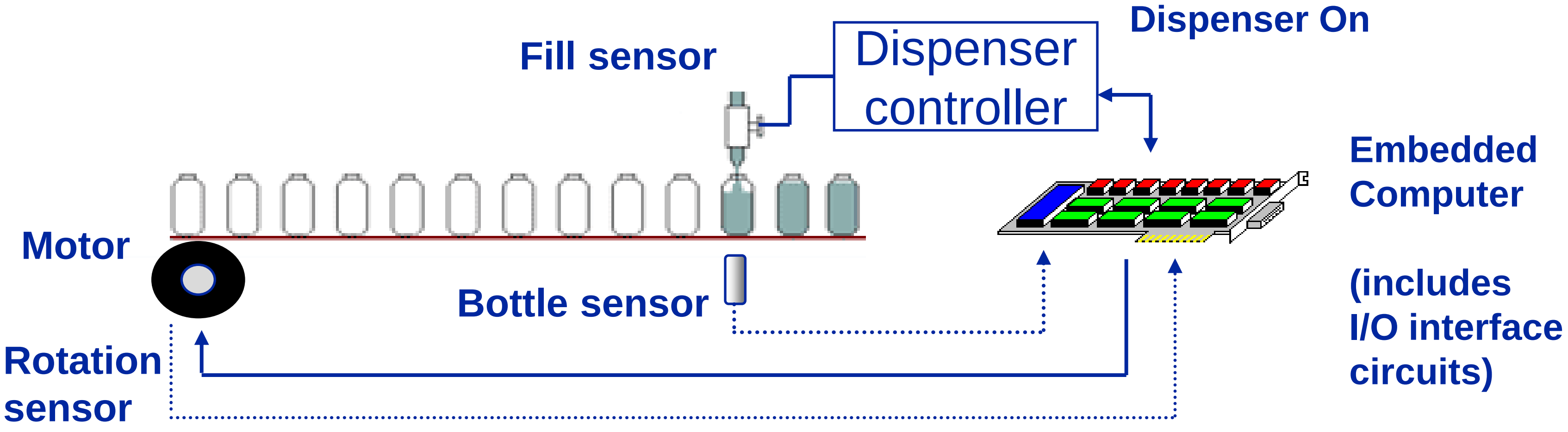
SET

TICK event

State machine: wristwatch



State machines: logic control



Fill sensor

Dispenser controller

Dispenser On

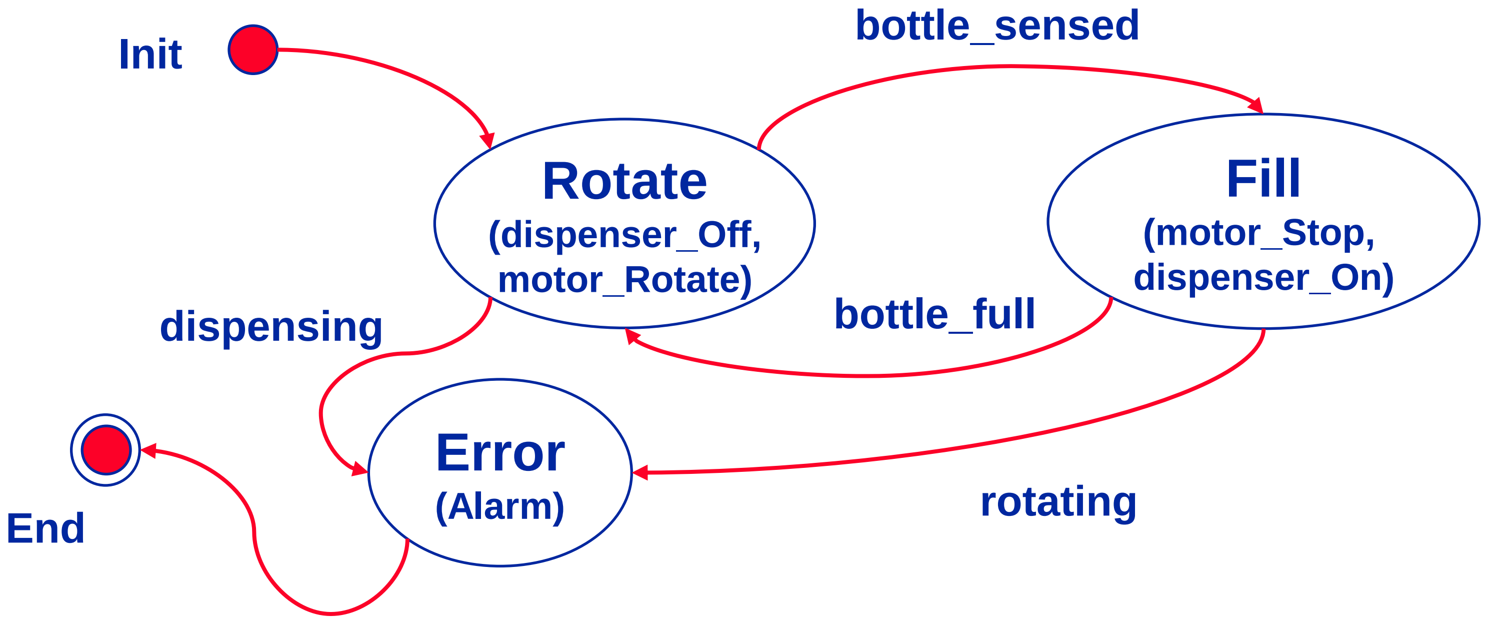
Embedded Computer

(includes I/O interface circuits)

Motor

Bottle sensor

Rotation sensor



Init

bottle_sensed

Rotate (dispenser_Off, motor_Rotate)

Fill (motor_Stop, dispenser_On)

bottle_full

dispensing

Error (Alarm)

rotating

End

State Machine Programming Example

```
enum State {Rotate, Fill,
            Error};
enum Event {bottle_sensed,
            bottle_full,
            dispensing,
            rotating};

void motor_Stop();
void motor_Rotate();
void dispenser_On();
void dispenser_Off();
void Alarm();

motor_Rotate();
dispenser_Off();
static State s = Rotate;
          .
          .
          .
```

```
void Transition(Event e)
{ switch(s)
     {  case Rotate:
             switch(e)
             { case bottle_sensed:
                    s = Fill;
                    motor_Stop();
                    dispenser_On();
                    break;
               case dispensing:
                    s = Error;
                    Alarm();
                    break;
             } break;
        case Fill:
             switch(e)
             { case bottle_full:
                    s = Rotate;
                    dispenser_Off();
                    motor_Rotate();
                    break;
               case rotating:
                    s = Error;
                    Alarm();
                    break;
             } break;
        case Error:
               sleep();
               break;
     }
}
```

Event-Based Programming

- Canned CE/CS programming assignments:
    - Initialize
    - Compute Something
    - Print Something
    - Return
- Event-Based Programming
    - Initialize
    - While (!end_of_the_universe)
        - Check for events
        - Forward them to your state machine
            - Ex.: Transition() on last slide
        - Check for Errors
    - … should never get here

Outline

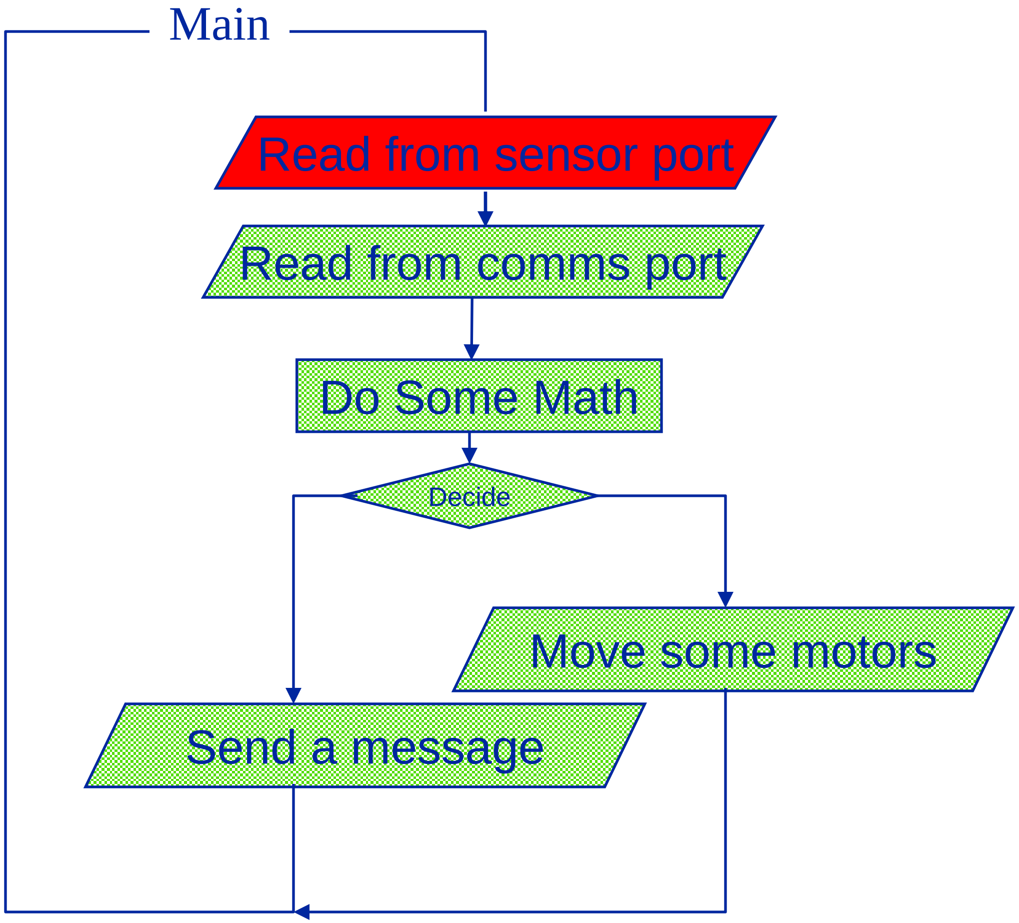
- Microcontroller Overview
- Event-Based Programming
  - State machines
  - Polling & Interrupts
- Microcontroller Digital I/O
  - On/off I/O
    - LEDs, solenoids, switches
  - Timers
  - Pulse-based I/O
    - PWM motor driving
    - Encoder position feedback
    - Serial and interprocessor communications

Catching Events

- Polling
  - Are we there yet?
    - Repeat continuously
  - Also known as busy-waiting
  - Sometimes this is the only option
    - If so, you should reevaluate your hardware/OS selection.
- Interrupts
  - Wake me up when we get there
    - Snooze or work on other things
  - Requires dedicated hardware
  - Better? Worse?
    - The answer is: usually better, but it depends…

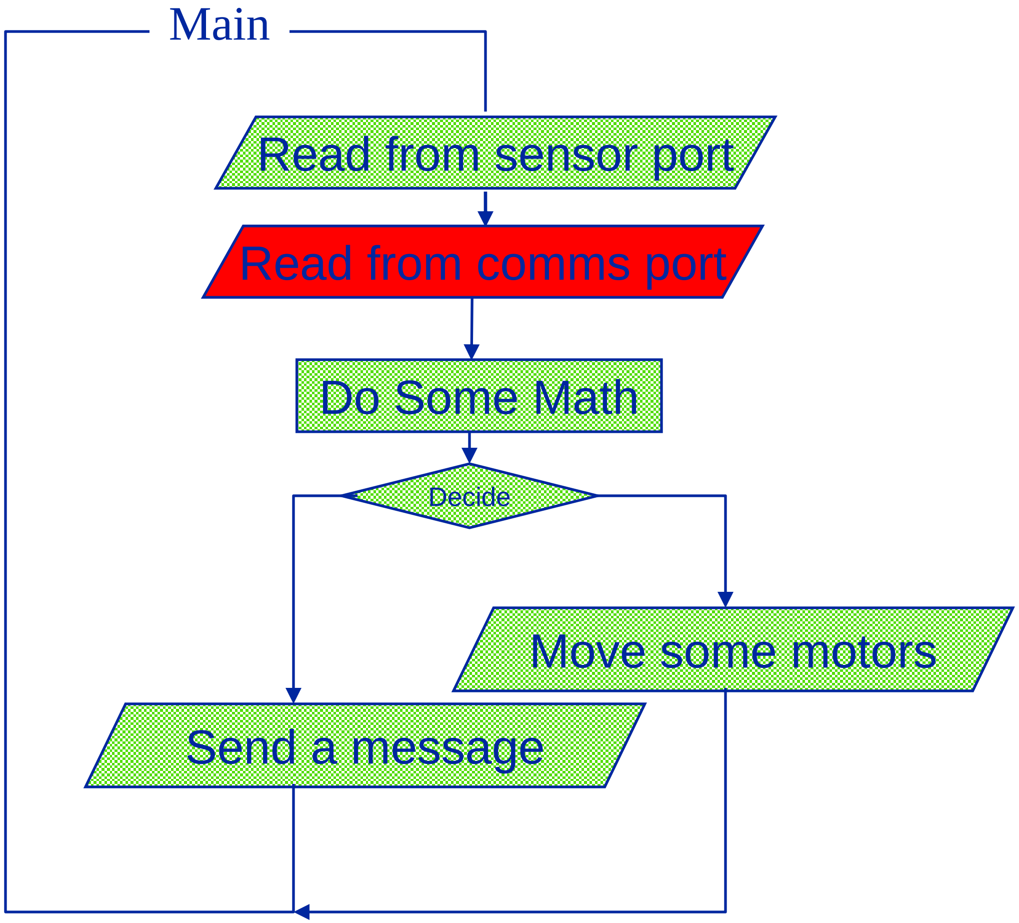
Polling

- **Single control flow**
  - **Ok for most applications**
  - **Sense-think-act paradigm**



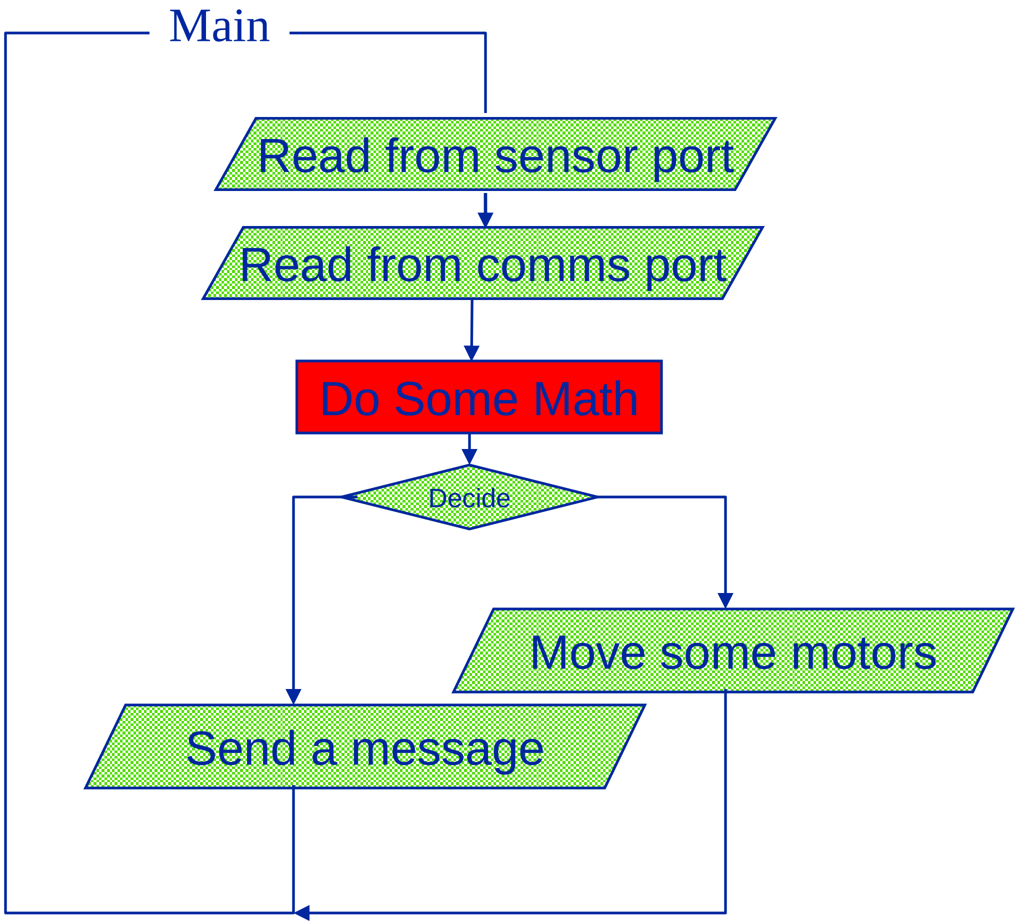
Polling

- **Single control flow**
  - **Ok for most applications**
  - **Sense-think-act paradigm**



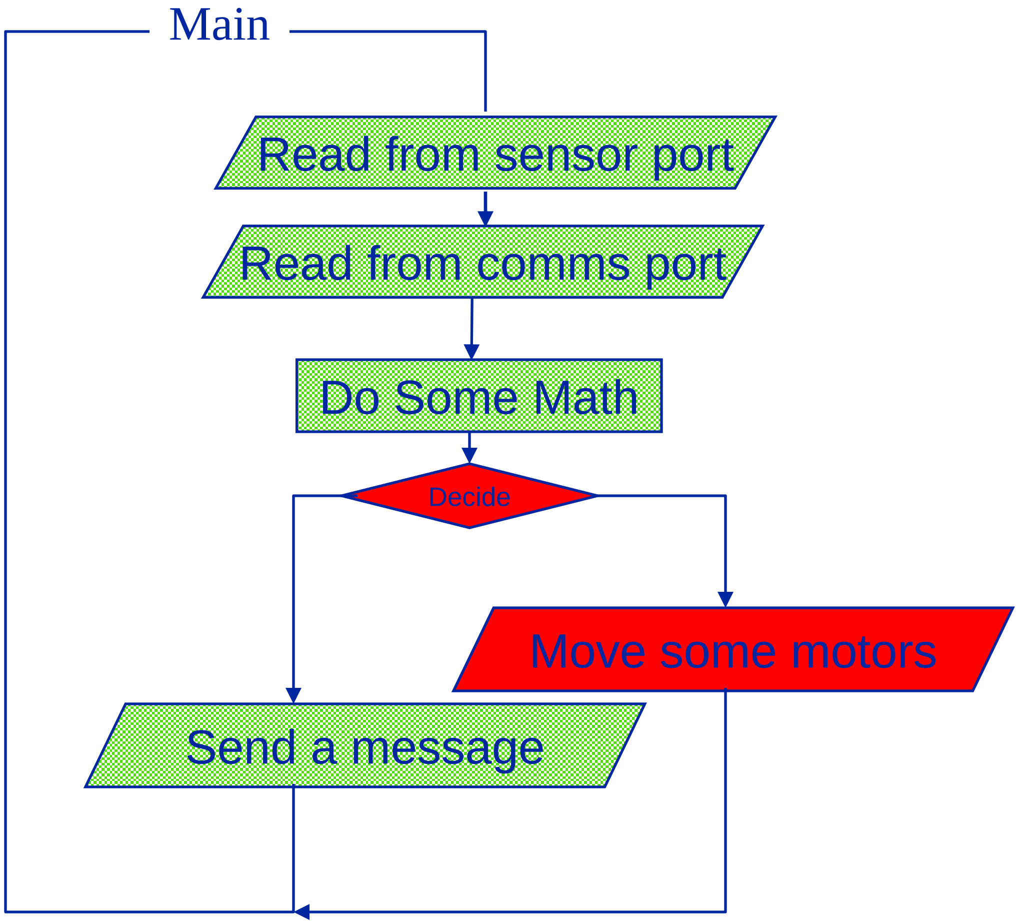
Polling

- **Single control flow**
    - **Ok for most applications**
    - **Sense-think-act paradigm**



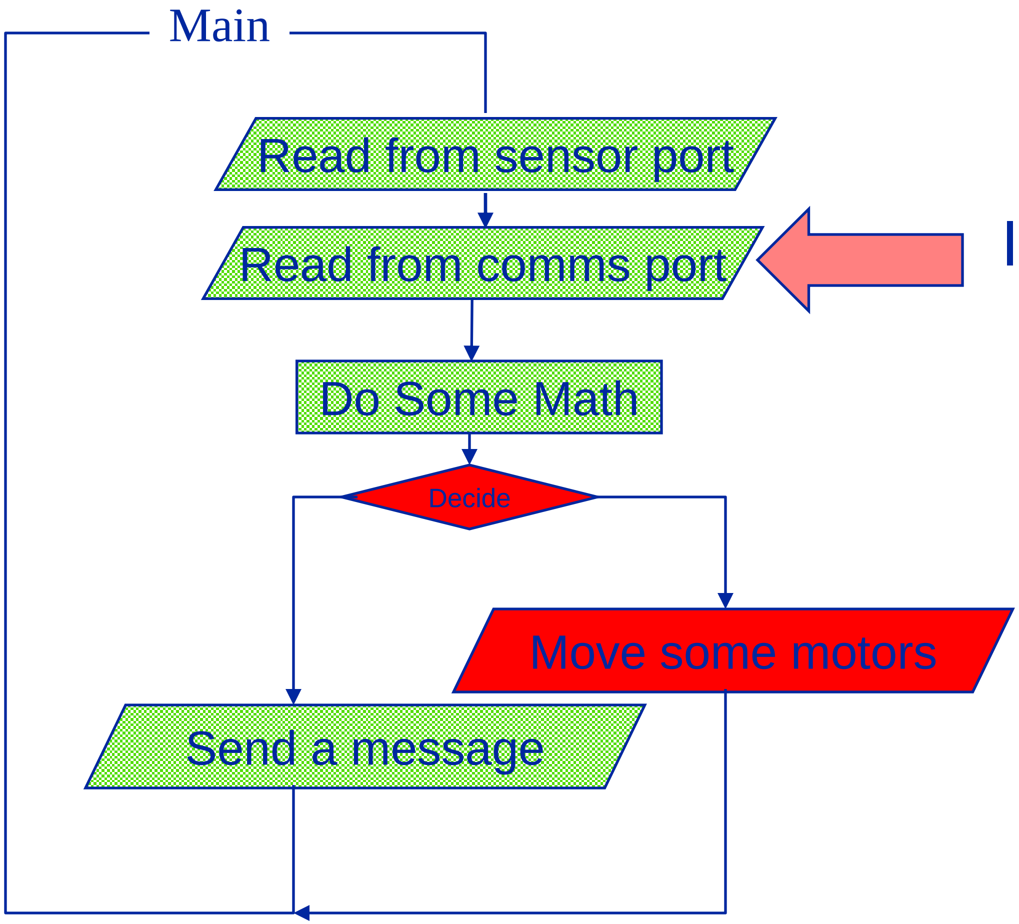
Polling

- **Single control flow**
  - **Ok for most applications**
  - **Sense-think-act paradigm**



Polling

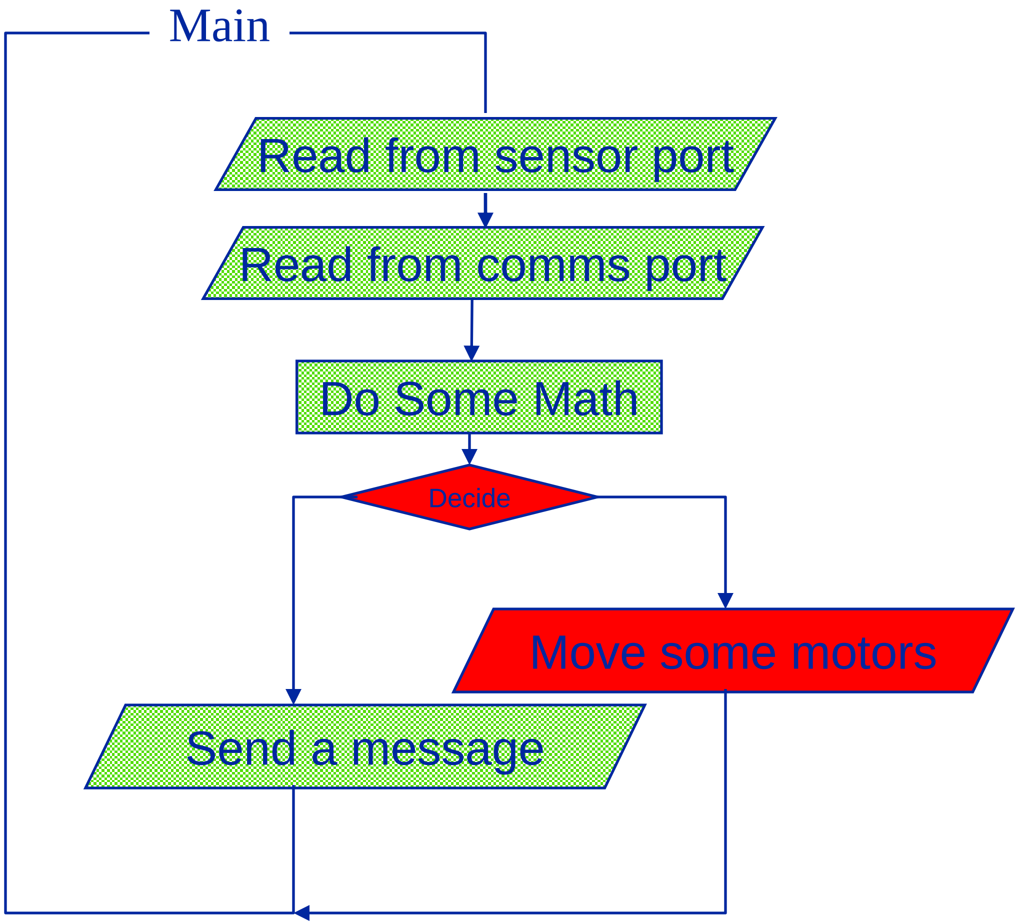
- **Single control flow**
  - **Ok for most applications**
  - **Sense-think-act paradigm**

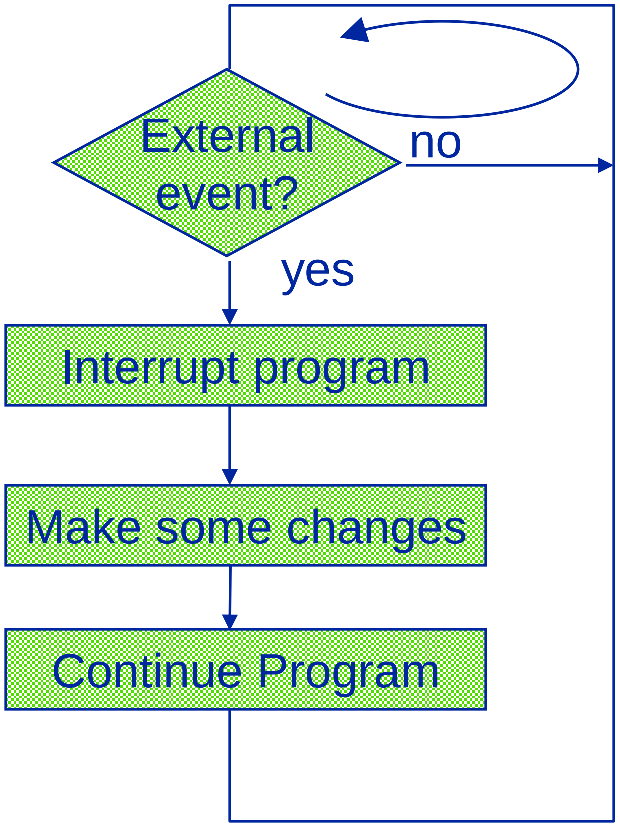


Important Message Missed!!

Interrupts

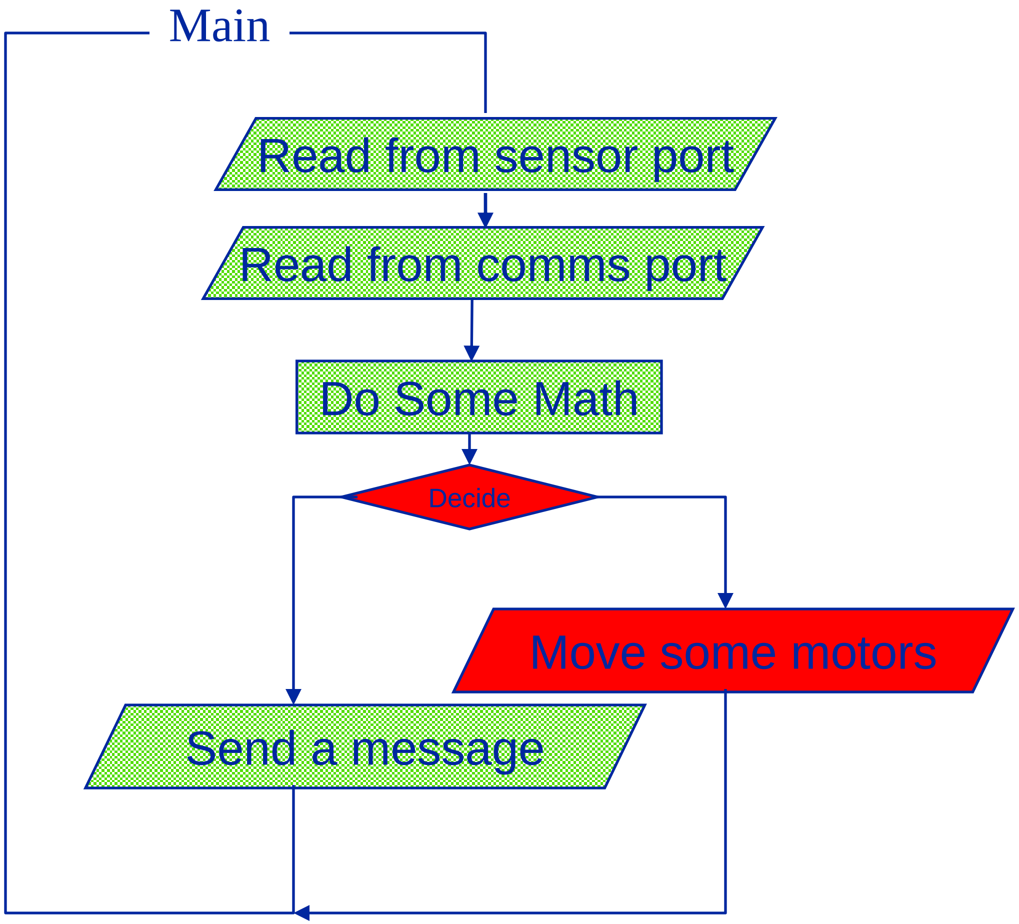
- **Single control flow**
  - **Ok for most applications**
  - **Sense-think-act paradigm**

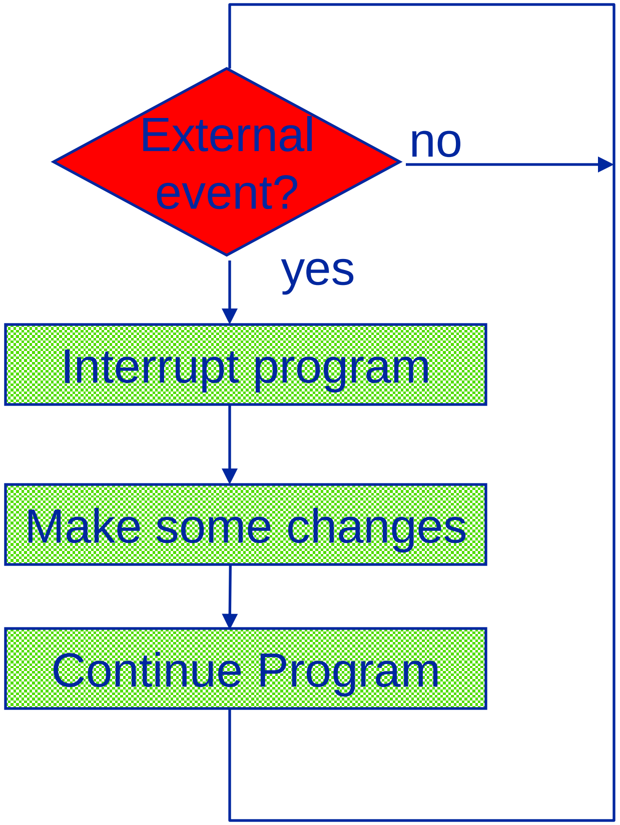




Interrupts

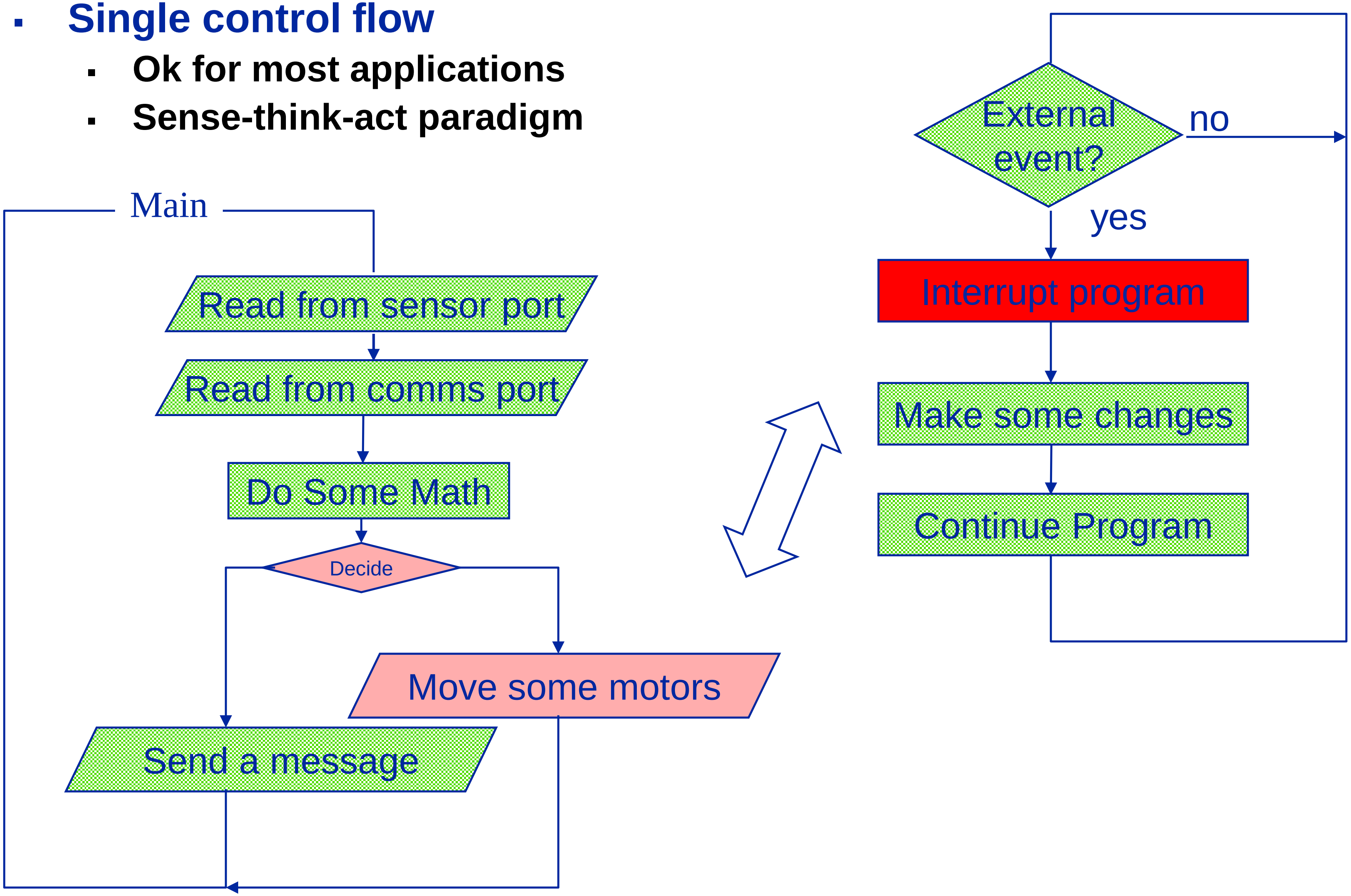
- **Single control flow**
  - **Ok for most applications**
  - **Sense-think-act paradigm**





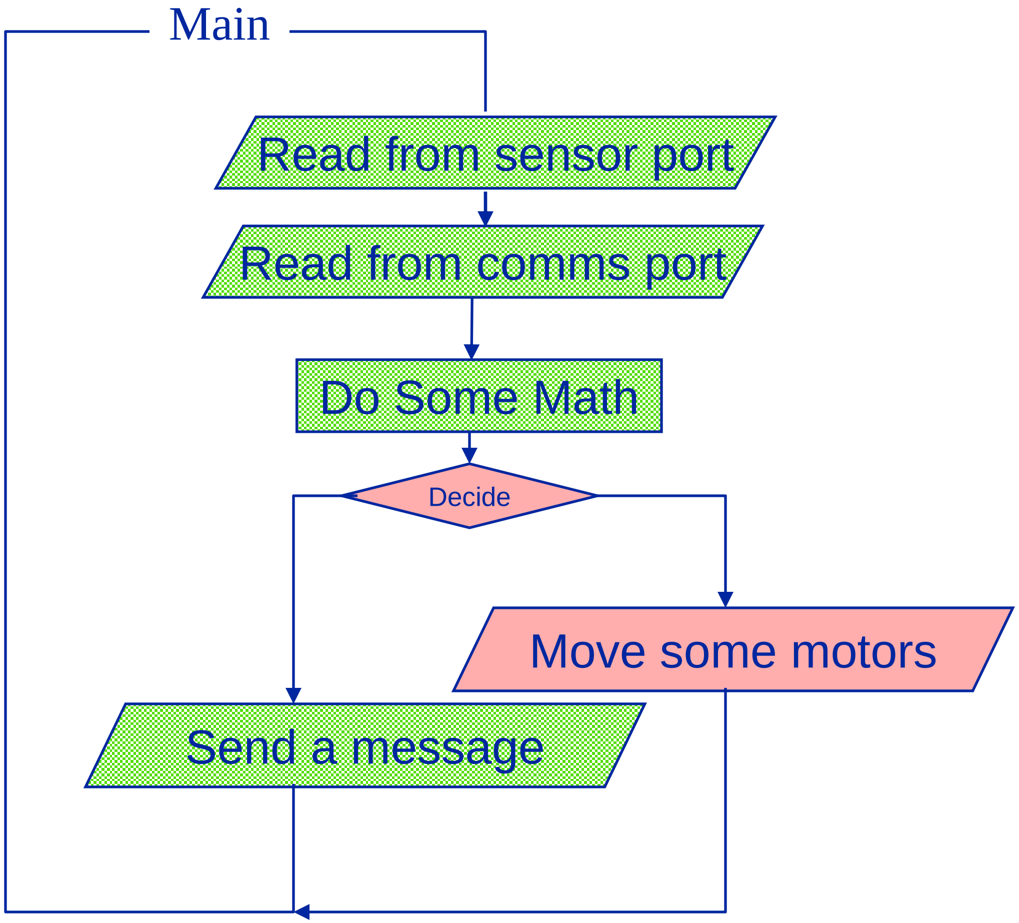
Interrupts

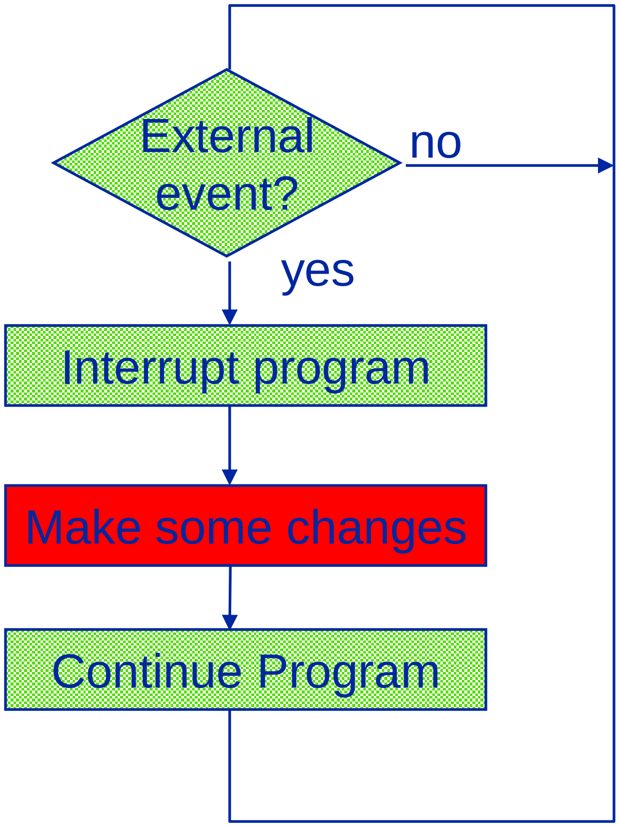
- **Single control flow**
  - **Ok for most applications**
  - **Sense-think-act paradigm**



Interrupts

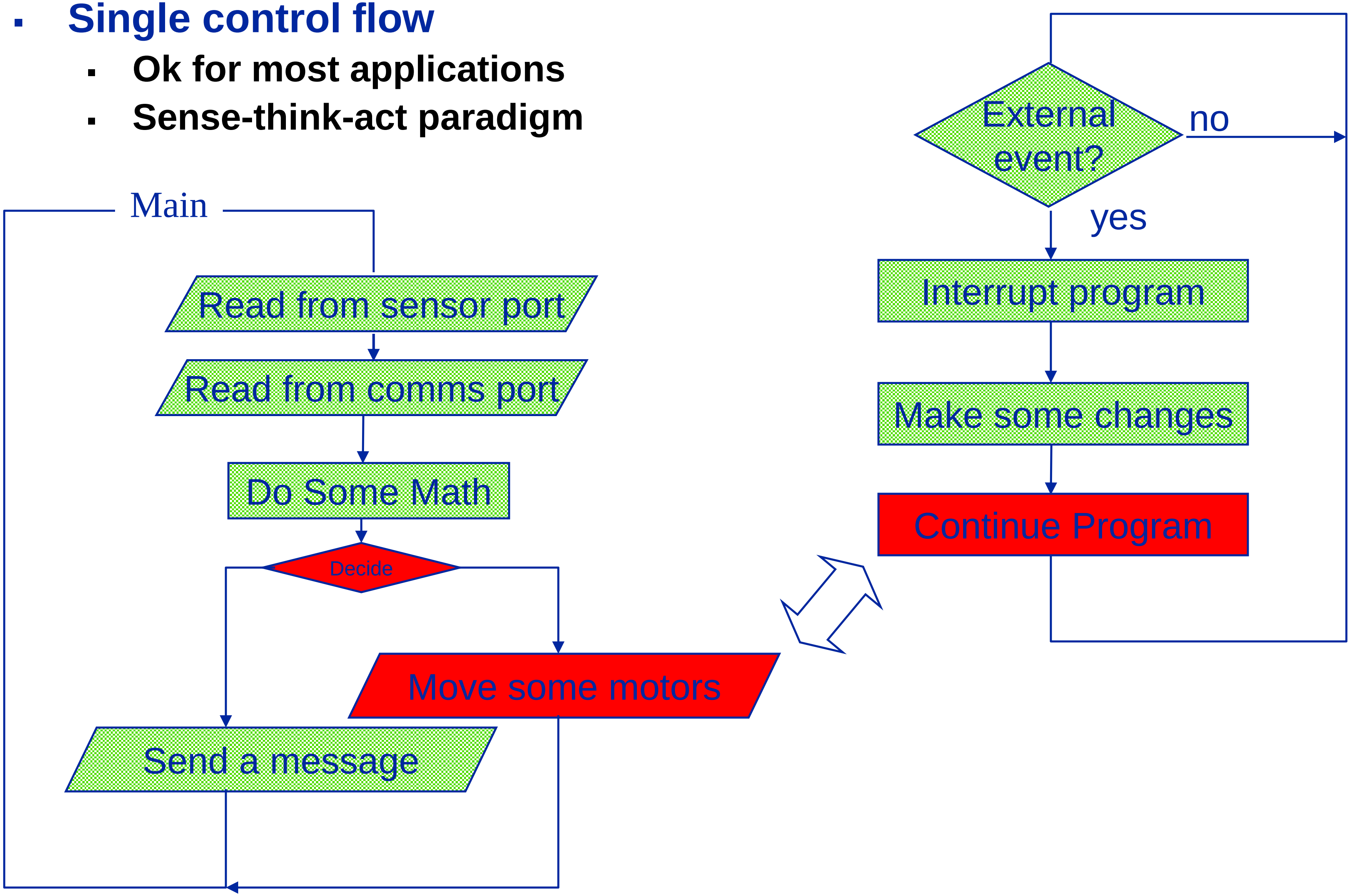
- **Single control flow**
  - **Ok for most applications**
  - **Sense-think-act paradigm**





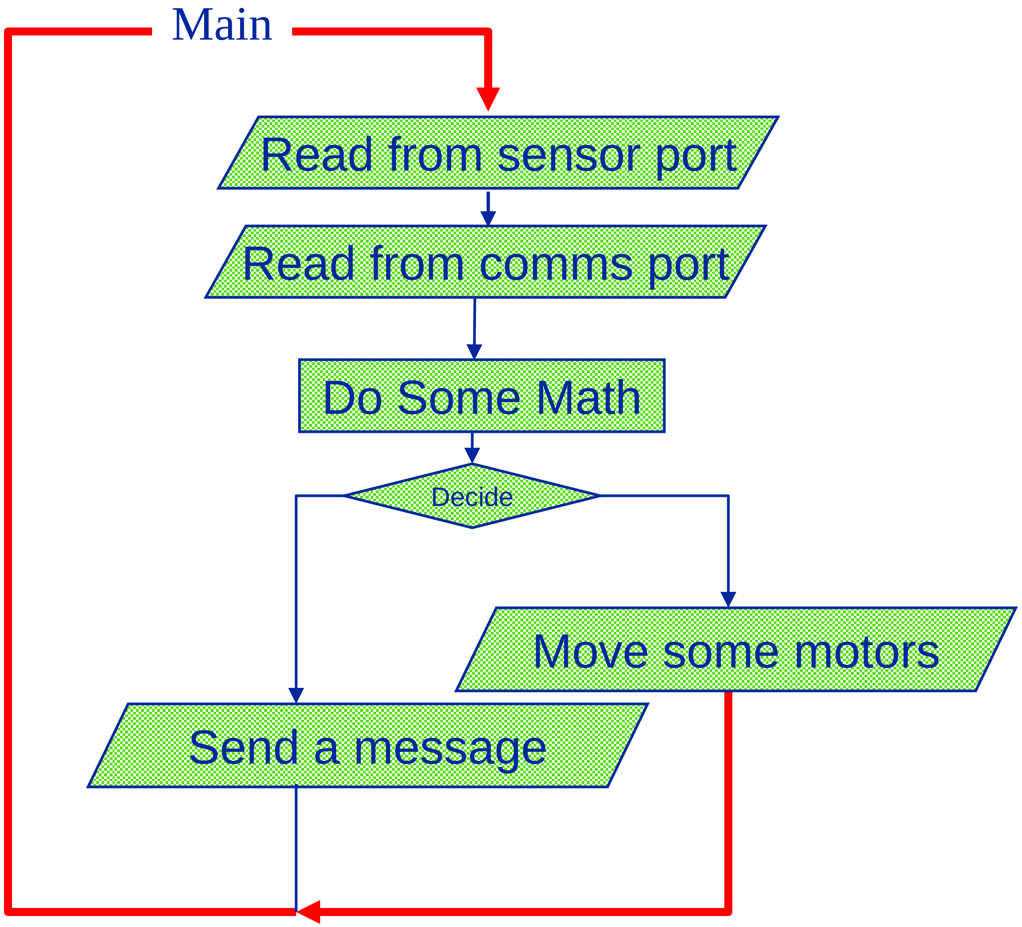
Interrupts

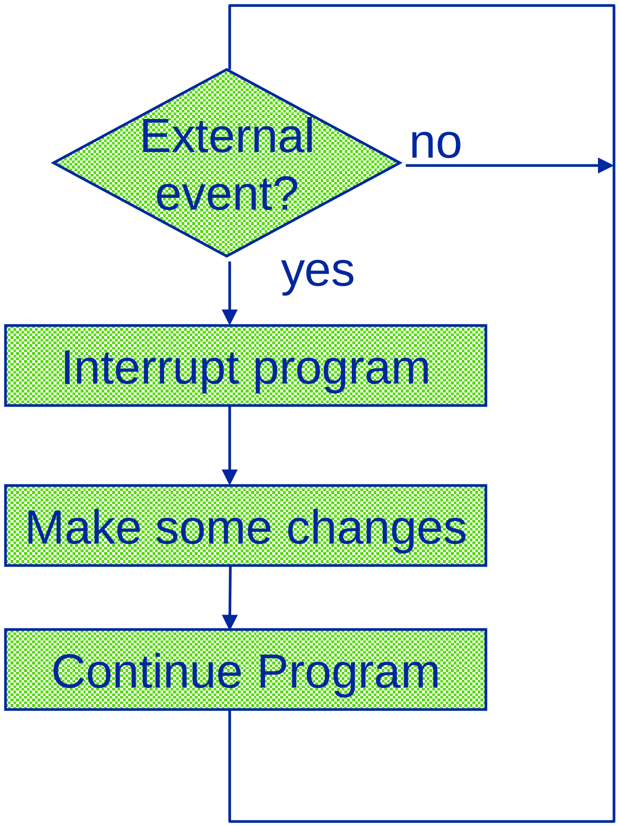
- **Single control flow**
  - **Ok for most applications**
  - **Sense-think-act paradigm**



Interrupts

- **Single control flow**
  - **Ok for most applications**
  - **Sense-think-act paradigm**





Example – Main Loop Using Polling

```
void main() {
float a;
do {
    for(a=1;a<=10;a=a+0.5)
      {
         printf("\r\nSqrt(%f) = %f", a, sqrt(a)); // Calc & print square root
         delay_ms(1000);
         if (kbhit()) {
            SerialData = getc();
            switch(SerialData)
            {  case '1': event = bottle_sensed;
                  Transition(event);
                  break;
               case '2': event = bottle_full;
                  Transition(event);
                  break;
               case '3': event = dispensing;
                  Transition(event);
                  break;
               case '4': event = rotating;
                  Transition(event);
                  break;
            }
          }
      }
   } while (TRUE);
}
```

Retrieve data from buffer using getc

kbhit tells us whether there is a new datum available

Example – Main Loop Using Interrupts

```
// Serial port interrupt service routine
void serial_isr() {
     SerialData = getc();
     switch(SerialData)
     {  case '1': event = bottle_sensed;
           Transition(event);
           break;
        case '2': event = bottle_full;
           Transition(event);
           break;
        case '3': event = dispensing;
           Transition(event);
           break;
        case '4': event = rotating;
           Transition(event);
           break;
     }
}
```

Retrieve data from buffer using `getc`

```
// Main routine
void main() {
float a;

enable_interrupts(…);
do {
    for(a=1;a<=10;a=a+0.5)
    {
       printf("\r\nSqrt(%f) = %f",a,sqrt(a));
       delay_ms(1000);
    }
   } while (TRUE);
}
```

Arduino Example: Main Loop Using Polling

```
// Program that counts the number of state changes on an input pin
#define inputPin 9
int stateChanges;         // Counts the number of state changes
boolean oldValue, newValue;

void setup() {
        pinMode(inputPin, INPUT);          // Set inputPin to be digital input
        digitalWrite(inputPin, HIGH);      // Enable 20KΩ internal pull-up resistor
        oldValue = digitalRead(inputPin); // Read inputPin's initial value
        stateChanges = 0;                  // No state changes yet
        Serial.begin(9600);        // Open serial port, set data rate to 9600 Baud
}

void loop() {
        newValue = digitalRead(inputPin);  // Read inputPin's new value
        if (newValue != oldValue){         // If the input has changed...
                oldValue = newValue;
                stateChanges = stateChanges + 1;
                Serial.println(stateChanges);
        }

        coolStuff(); // Main things your program does
}
```

Arduino Example: Main Loop Using Interrupts

```
// Program that counts the number of state changes on an input pin
#define inputPin 2 // Associated with external interrupt 0 on the Arduino
volatile int stateChanges;

void setup() {
        pinMode(inputPin, INPUT); // Set inputPin to be digital input
        digitalWrite(inputPin, HIGH); // Enable 20KΩ internal pull-up resistor

        // Now we want to attach the interrupt for a change on the pin
        attachInterrupt(0, inputChanged, CHANGE); // Could be LOW, RISING, FALLING
        // See: http://www.arduino.cc/en/Reference/AttachInterrupt

        stateChanges = 0;   // No state changes yet
        Serial.begin(9600); // Opens serial port, sets data rate to 9600 Baud
}

void inputChanged() {       // Interrupt Service Routine (ISR)...keep it simple!
        stateChanges = stateChanges + 1;   // Increment the # of state changes

        Serial.println(stateChanges);      // Print the # of state changes
}

void loop() {
        coolStuff(); // Main things your program does
}
```

Outline

- Microcontroller Overview
- Event-Based Programming
    - State machines
    - Polling & Interrupts
- Microcontroller Digital I/O
    - On/off I/O
        - LEDs, solenoids, switches
    - Timers
    - Pulse-based I/O
        - PWM motor driving
        - Encoder position feedback
        - Serial and interprocessor communications

Microcontrollers: Digital I/O

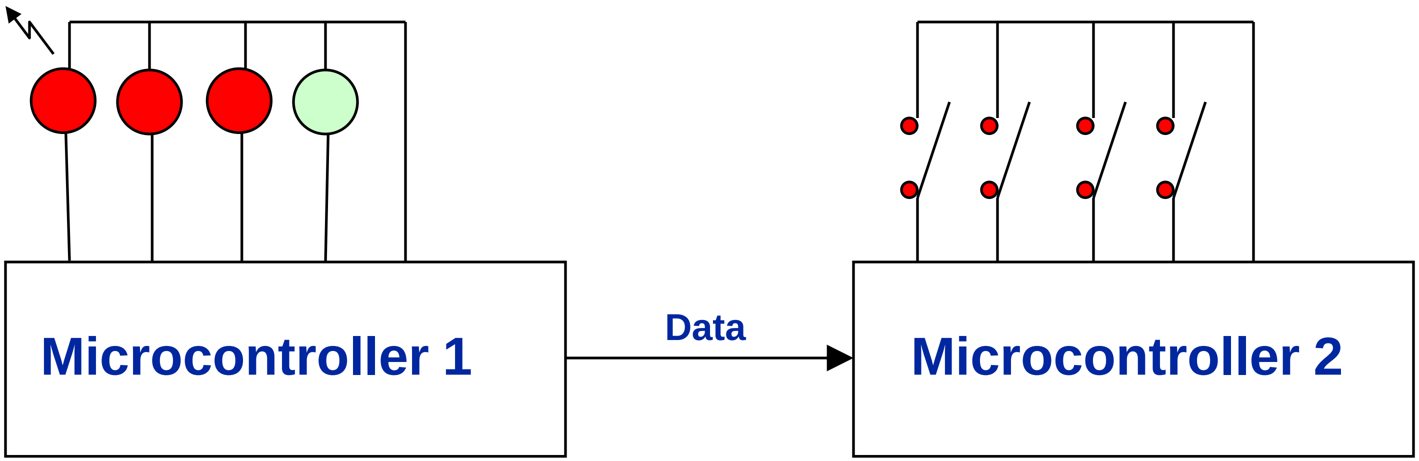
- **Logic Output**
  - **Light LEDs, trigger solenoids**
  - **Motor control**
  - **Send data**

ON/OFF

Pulse generation

Pulse interpretation

- **Logic Input**
  - **Monitor switches**
  - **Monitor sensor logic**
  - **Receive data**



Digital I/O Preliminaries

- All microcontroller peripherals are configured or manipulated via *control* and *status* registers
- Data Direction Register (a *control* register)
  - Associated with a (typically 8-bit) port
  - Writing to it sets the pins to inputs or outputs
  - Different names (Atmel: DDR, TI: Direction Register, Microchip: TRIS), but same function
- Input/Output Register (a *status* register)
  - Reads from or writes to a port
  - Two approaches: separate register for input and output, or a combined register for both
  - Combined register selectively writes only to the designated output pins, reads only from the designated input pins on a given port

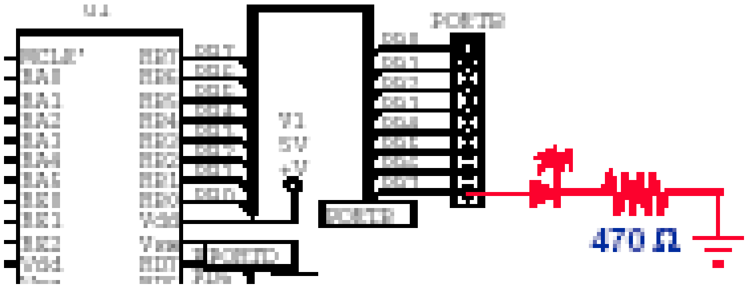
On/Off Logic Output: Flashing LED

#include and other pre-compiler statements (none needed here)

```
void setup() {                  // Runs once
  // initialize the digital pin as an output
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(13, OUTPUT);
}
```

```
void loop() {                    // Repeats
  digitalWrite(13, HIGH);        // set the LED on
  delay(1000);                   // wait for a second
  digitalWrite(13, LOW);         // set the LED off
  delay(1000);                   // wait for a second
} // http://arduino.cc/en/Tutorial/Blink
```

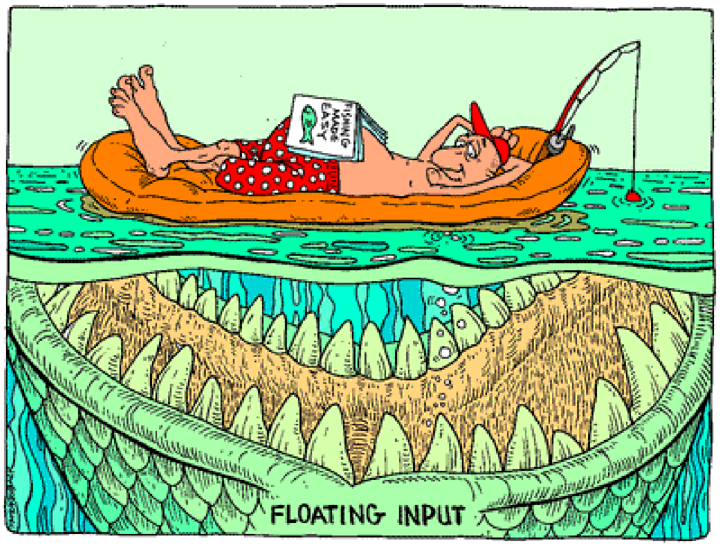
main program body

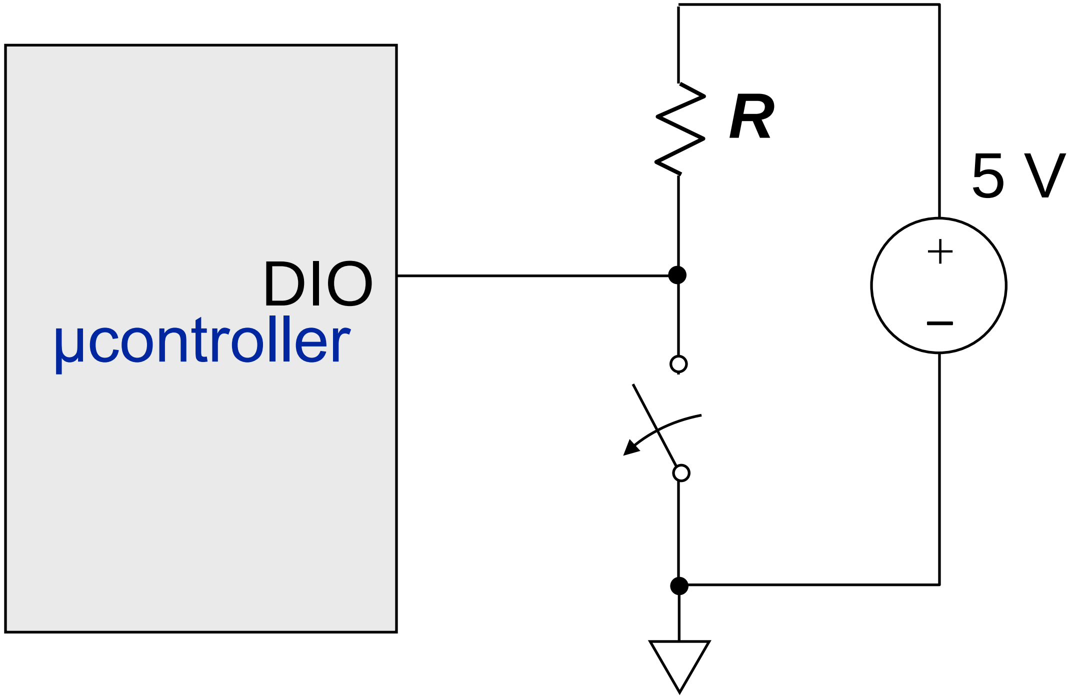


software is closely related to hardware

On/Off Logic Input: Switch Connection

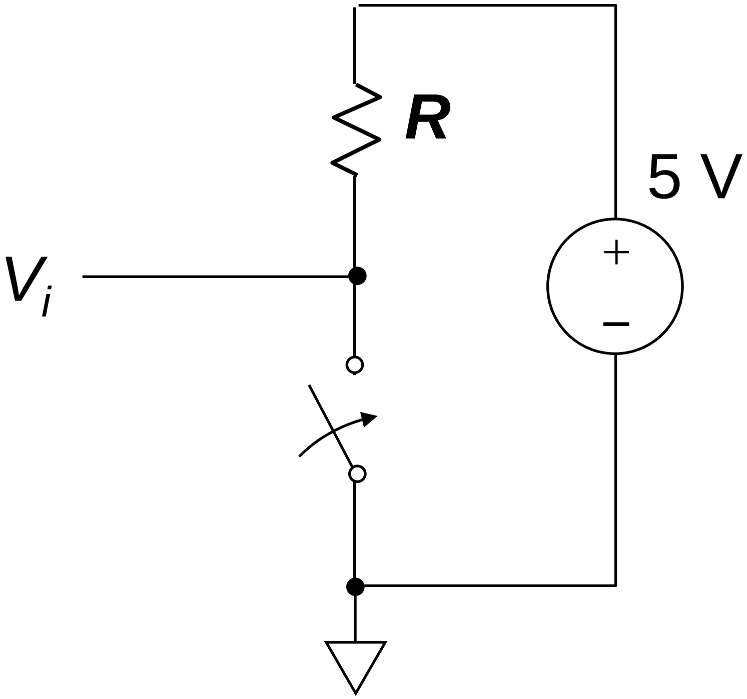
- Switch connection to µcontroller without resistor: will it work?
- Need “pull-up” resistor
  - Set $R \geq 10$ kΩ to limit current to ≤ 0.5 mA for switch closed
  - Arduino has internal pull-up resistor that can be enabled…
- Can swap role of switch and resistor, if desired

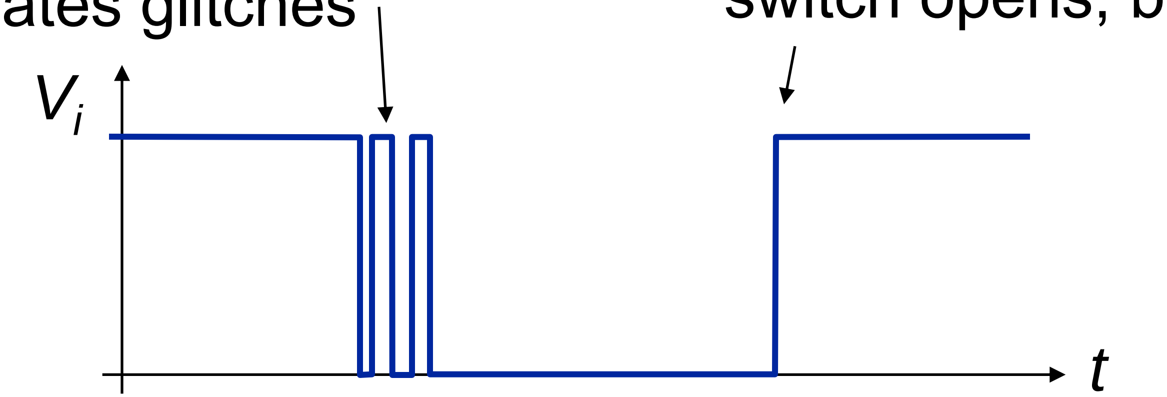




On/Off Logic Input: Switch Bounce

- **Problem: Signal from switch is not really “digital”**
- **Usually switch “bounces”**
  - **Switch temporarily becomes open-circuit**
  - **~Millisecond bouncing**
- **Program may think switch was pressed multiple times**





Outline

- Microcontroller Overview
- Event-Based Programming
  - State machines
  - Polling & Interrupts
- Microcontroller Digital I/O
  - On/off I/O
    - LEDs, solenoids, switches
  - Timers, output compare, input capture, interrupts
  - Pulse-based I/O
    - PWM motor driving
    - Encoder position feedback
    - Serial and interprocessor communications

Microcontroller Clocks

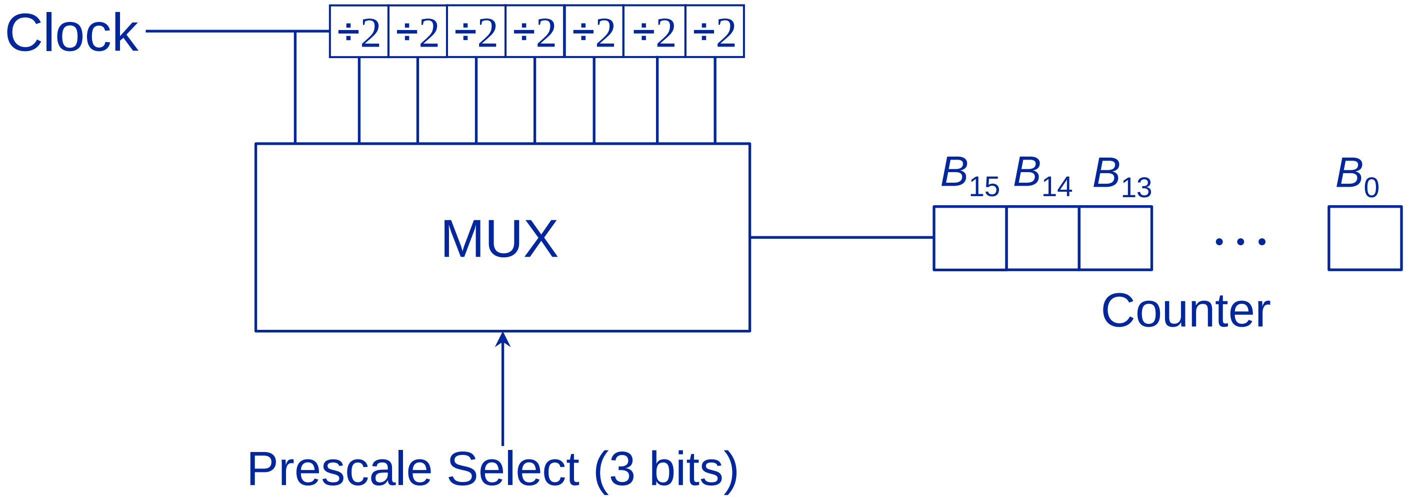
- Synchronizes the microcontroller’s fetch-execute cycle
- Internal clock
  - Comes with the microcontroller, typically RC-based
  - No external components needed, freeing pins
  - Usually less accurate than external oscillator
- External clock
  - Crystal + capacitors
  - Ceramic resonator + capacitors
  - RC (including silicon oscillator)
  - “Oscillator module” integrates all components into single package

Timers/(Counters)

- Dedicated hardware for measuring the passage of time or counting events
- Common microcontroller timer systems
  - Timer overflow
  - Output compare
  - Input capture

Timer Basics

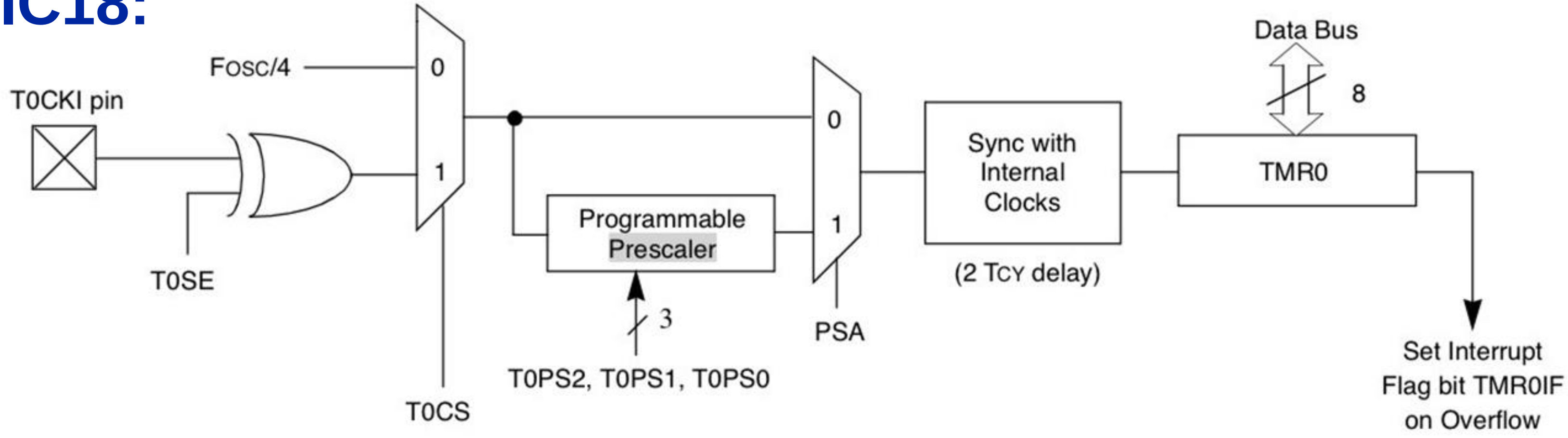
- A timer is typically an (event) counter that is agnostic about its clock source – can be internal or external, synchronous or asynchronous
- A timer often has an optional divider or "prescaler":



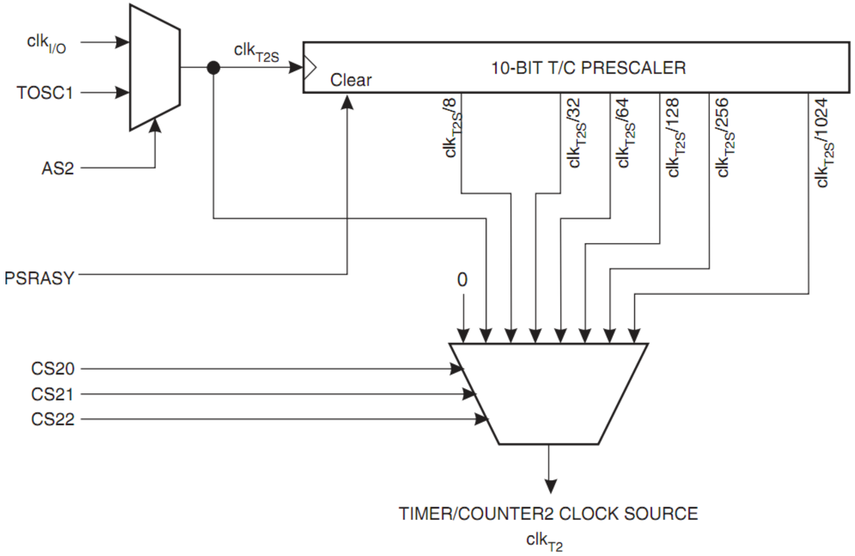
- Categories
  - Resettable: can be stopped or cleared at any time
  - Free-running: runs continuously, maybe with reload value

Timer Basics: Timer/Counter Examples (w/ Prescaler)

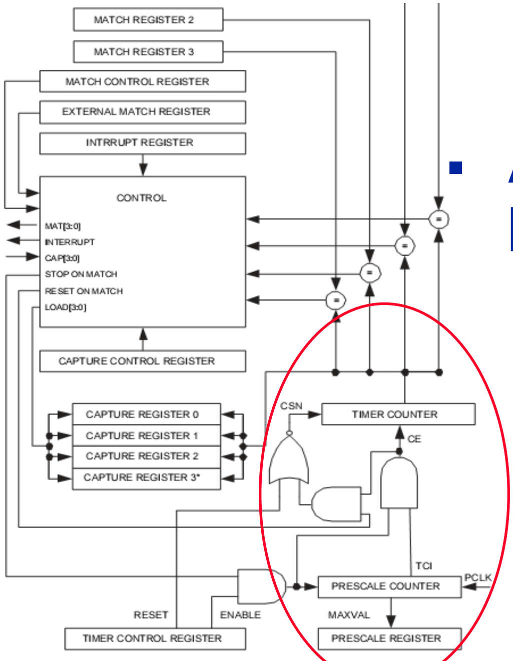
- **PIC18:**



- **ATmega328:**



- **ARM LPC2148**



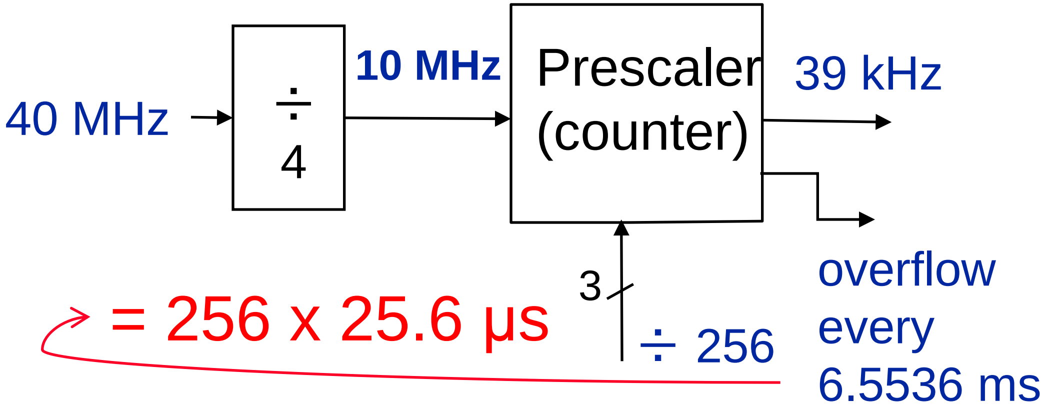
* Note: that the capture register 3 cannot be used on TIMER0

Timer Overflow

- No matter how big your timer/counter is, it will eventually “roll over”, or “overflow” back to 0
- This is typically indicated by a bit in a status register, and often by an interrupt
- Action on overflow is typically to reset the status bit to get ready for the next overflow
- A common use of overflow is to extend the length of time that can be measured by the timer

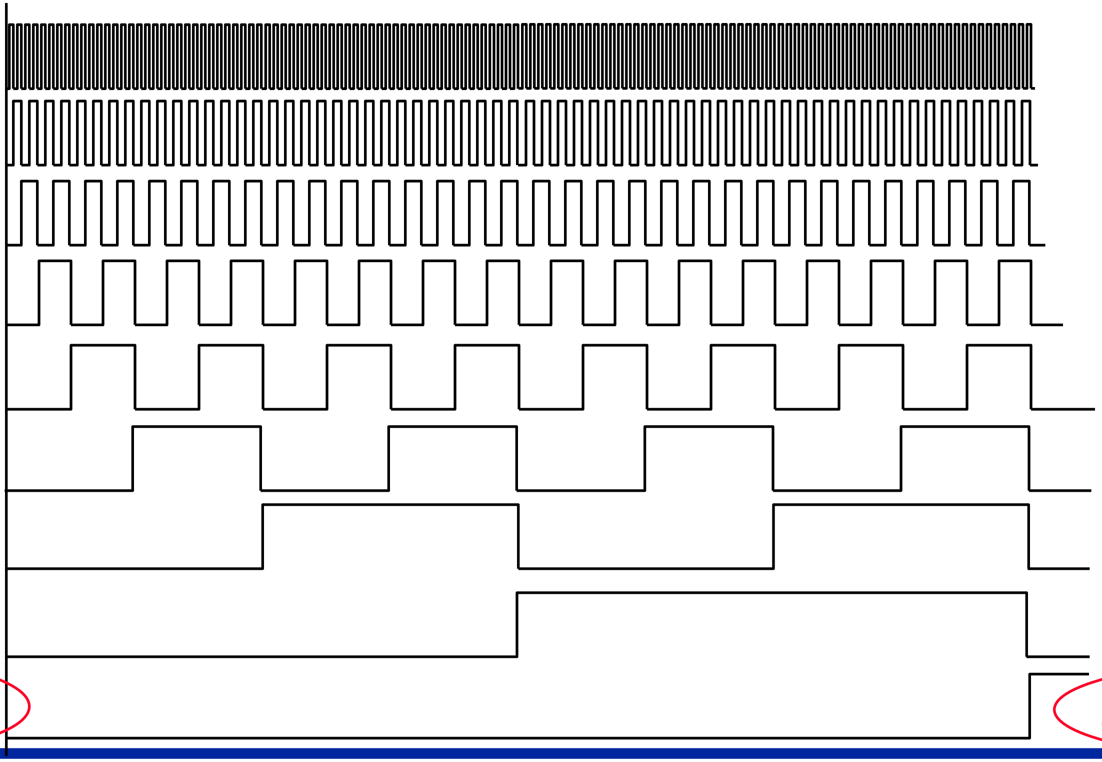
Timer Overflow Example (PIC18)

- **Prescaler divides clock down by binary fraction**
- **Useful for “real world” timing in seconds, minutes…**
- **Overflow at 256 counts**



overflow every 6.5536 ms

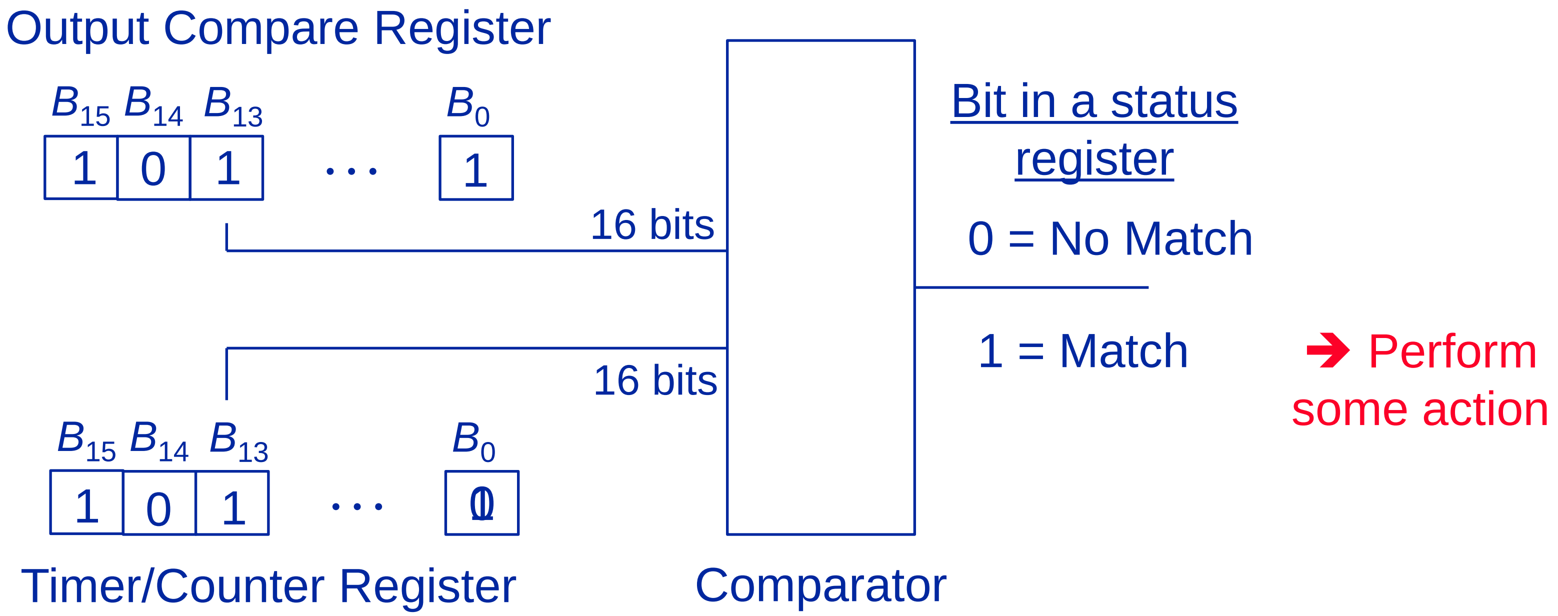
= 256 x 25.6 µs



Core Clock		0.1 µs	10 MHz
000	÷ 2	0.2 µs	5 MHz
001	÷ 4	0.4 µs	2.5 MHz
010	÷ 8	0.8 µs	1.25 MHz
011	÷ 16	1.6 µs	625 kHz
100	÷ 32	3.2 µs	312.5 kHz
101	÷ 64	6.4 µs	156.25 kHz
110	÷ 128	12.8 µs	78.125 kHz
111	÷ 256	25.6 µs	39062.5 Hz

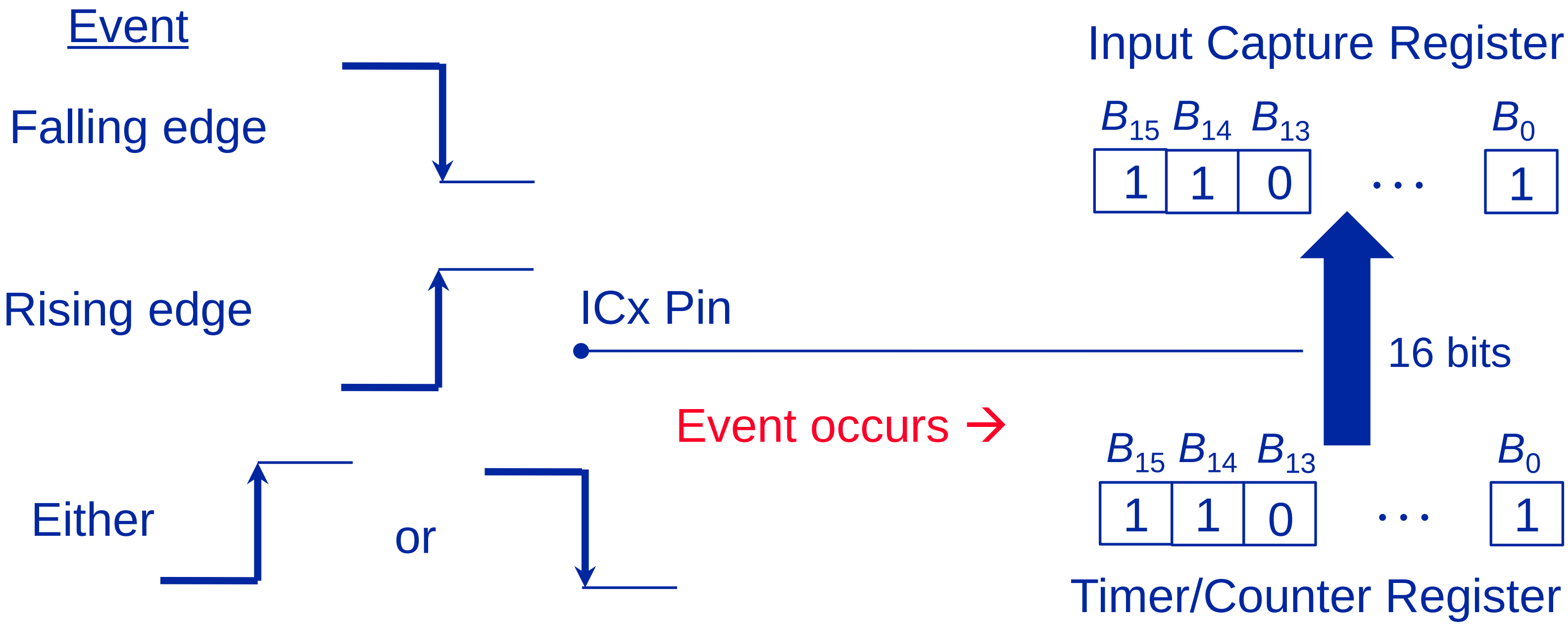
Output Compare

- **Detects when the value of a timer/counter is the same as a value that you set and store**
- **Example uses**
  - **Detect when some period of time has passed**
  - **Detect when a certain number of events has occurred**



Input Capture

- **Detects when an event happens and grabs the value of the counter at that instant**
- **Example uses**
  - **Detect how long it takes for something to happen**
  - **Detect time elapsed between events**



Summary: Output Compare & Input Capture

- Output Compare: generates an event when a pre-specified count is reached
    - Typical use: Generate a pulse train, e.g., to drive a motor
- Input Capture: generates a count when an event occurs
    - Typical use: Read an encoder, e.g., for motor position feedback

Interrupts

- Hardware interrupt: asynchronous signal indicating the need for attention
- An act of interrupting is an interrupt request (IRQ)
- Initiates execution of an interrupt handler or interrupt service routine (ISR)
- Maskable vs. non-maskable
  - Maskable: may be ignored via bit in interrupt mask register
  - Non-maskable: can’t be ignored (e.g., watchdog timer)
- Level-triggered vs. edge-triggered
  - Level-triggered: triggered by high or low logic level
  - Edge-triggered: triggered by a level transition

Interrupt Service Routines (ISR)

- Identify the routine as an ISR typically using a compiler directive (`#INT_xxx` for the PIC CCS compiler) or an “attachment” function or ISR macro (Arduino)
- The compiler will generate code that
  - jumps to the ISR when the interrupt is detected
  - disables interrupts
  - saves the machine state
  - runs your ISR code
  - restores the machine state
  - clears the interrupt
  - enables interrupts
  - jumps back to the main program

```
#INT_xxx
void ISR_NAME()
{
  // do something
}
```

Enabling and Assigning ISRs (Arduino)

- Global enabling of interrupts
  - `interrupts();`
- Global disabling of interrupts
  - `noInterrupts();`
- Assignment of individual interrupt
  - `attachInterrupt(interrupt#, function, mode);` OR
  - `ISR(Vector, Attributes); (ISR macro)`
- De-assignment of individual interrupt
  - `detachInterrupt(interrupt#);`

Interrupt Functional Types

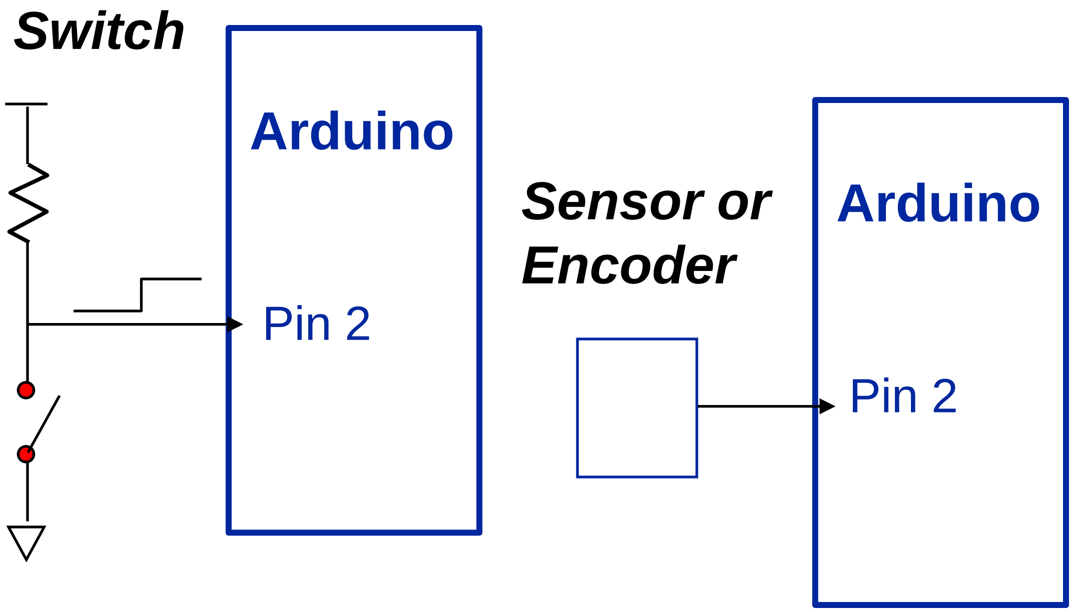
- External Interrupts
  - Single-pin change (Arduino: INT 0/1 on pins 2/3; Arduino Mega: additionally INT 2/3/4/5 on pins 21/20/19/18)
- "Input Change" or "Pin Change" Interrupts
  - Grouped pin change (PIC18: Port B, Pins 4:7)
  - Arduino PinChangeInt library (http://code.google.com/p/arduino-pinchangeint/)
- Timer Interrupts
  - Overflow (timer/counter, pre- or post-scaler)
- Compare or Capture Interrupts
  - Motor control or explicit timing
- Serial & Other Communications (e.g. I^{2}C) Interrupts
- ADC Interrupts
- …and others

Arduino: Selected Interrupts

- External interrupts
  - Most useful interrupt, handles changes to pins 2 and 3 (on Uno).
  - Built-in support in the Arduino library: attachInterrupt()
  - See: http://www.arduino.cc/en/Reference/AttachInterrupt
- Serial interrupts
  - They're handled (i.e., abstracted away) for you in the built-in Serial library.
  - See: http://arduino.cc/en/Reference/Serial
- Timer interrupts
  - See datasheet, you'll need to do a bit of lower-level coding.
  - Google is your friend…you won't be the first to do it.
- Other interrupts
  - If you've worked with microcontrollers before, find them in the datasheet.

Arduino: External Interrupt on Pins 2 and 3

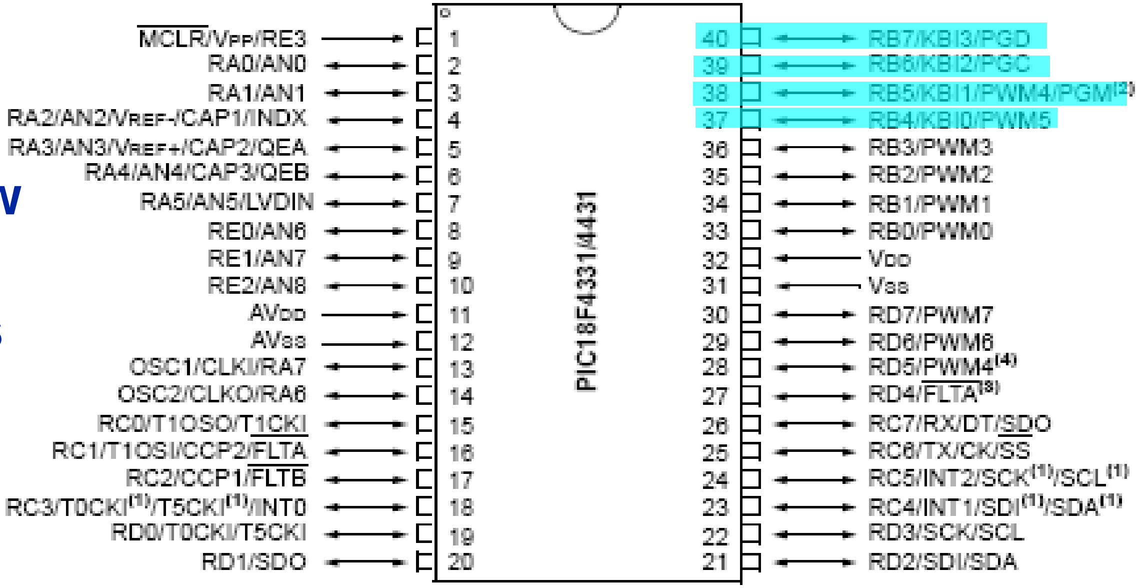
- Ideal for immediate action on pin voltage transition
- Can be set to detect a rising edge, falling edge, either edge, or low
- Will return to this…



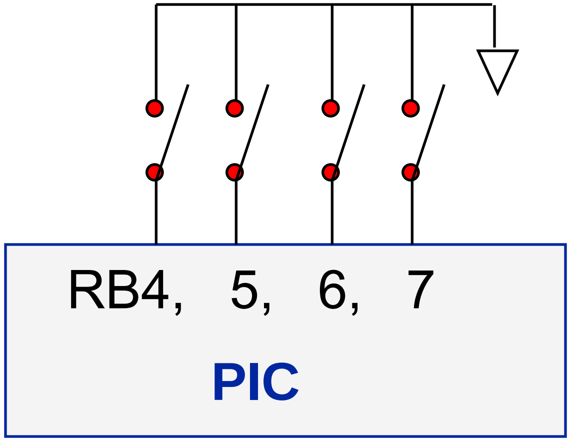
```
byte data;
void my_isr()
{
 // do something, e.g.,
 data = digitalRead(3);
}
```

PIC18: Event Interrupt - PORTB Change 4-7

- **Detect change on any of 4 pins, RB4-7**
- **Must read register to know which changed**
- **Good for reading keypads**
- **Must set pins to inputs**



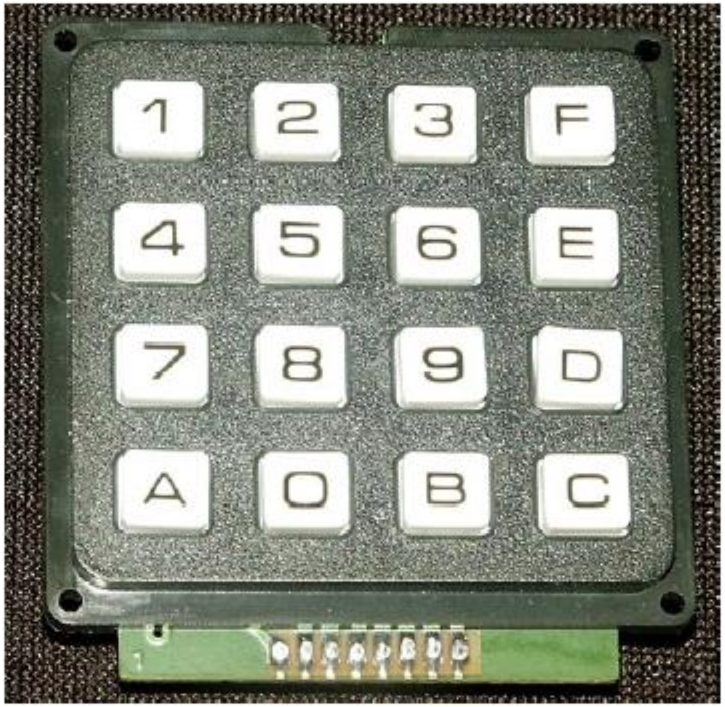
(pull-up resistors
not shown)



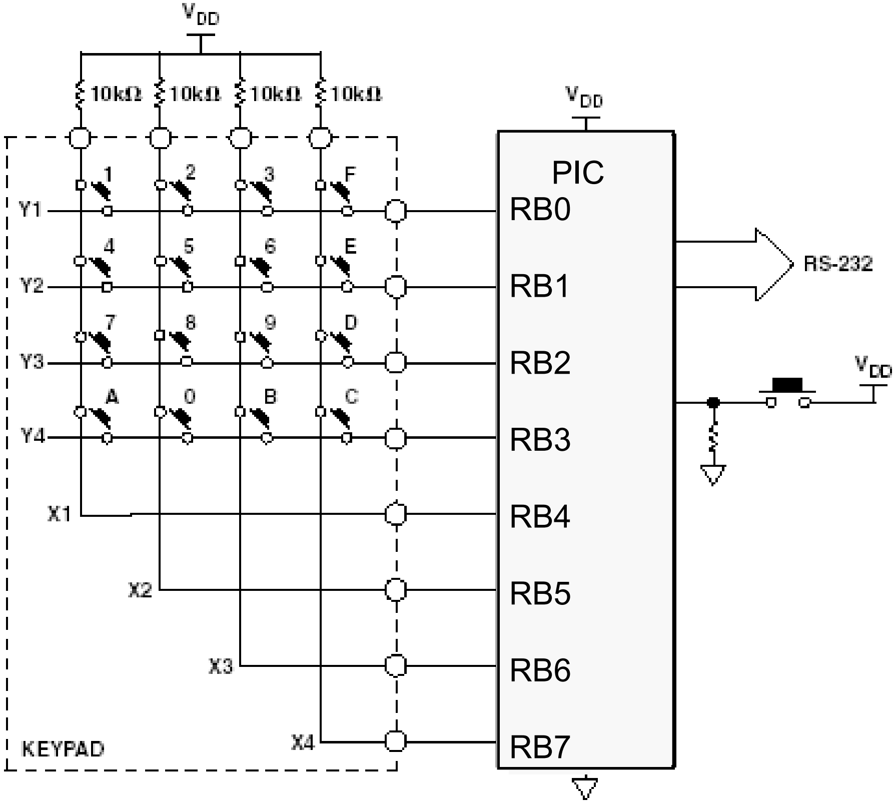
```
int val;
#INT_RB
void portb_isr()
{
   // Reading clears the interrupt
   val = input_b();
}
```

X-Y Keypads

- 8-pin interface (for 16 keys)
- 4 pins to RB4-7 for input (pins are high for no keypress)
- 4 pins to RB0-3 for polling column
  - all lines low for interrupt; poll lines for column



e.g. Grayhill Series 96 keypads $13



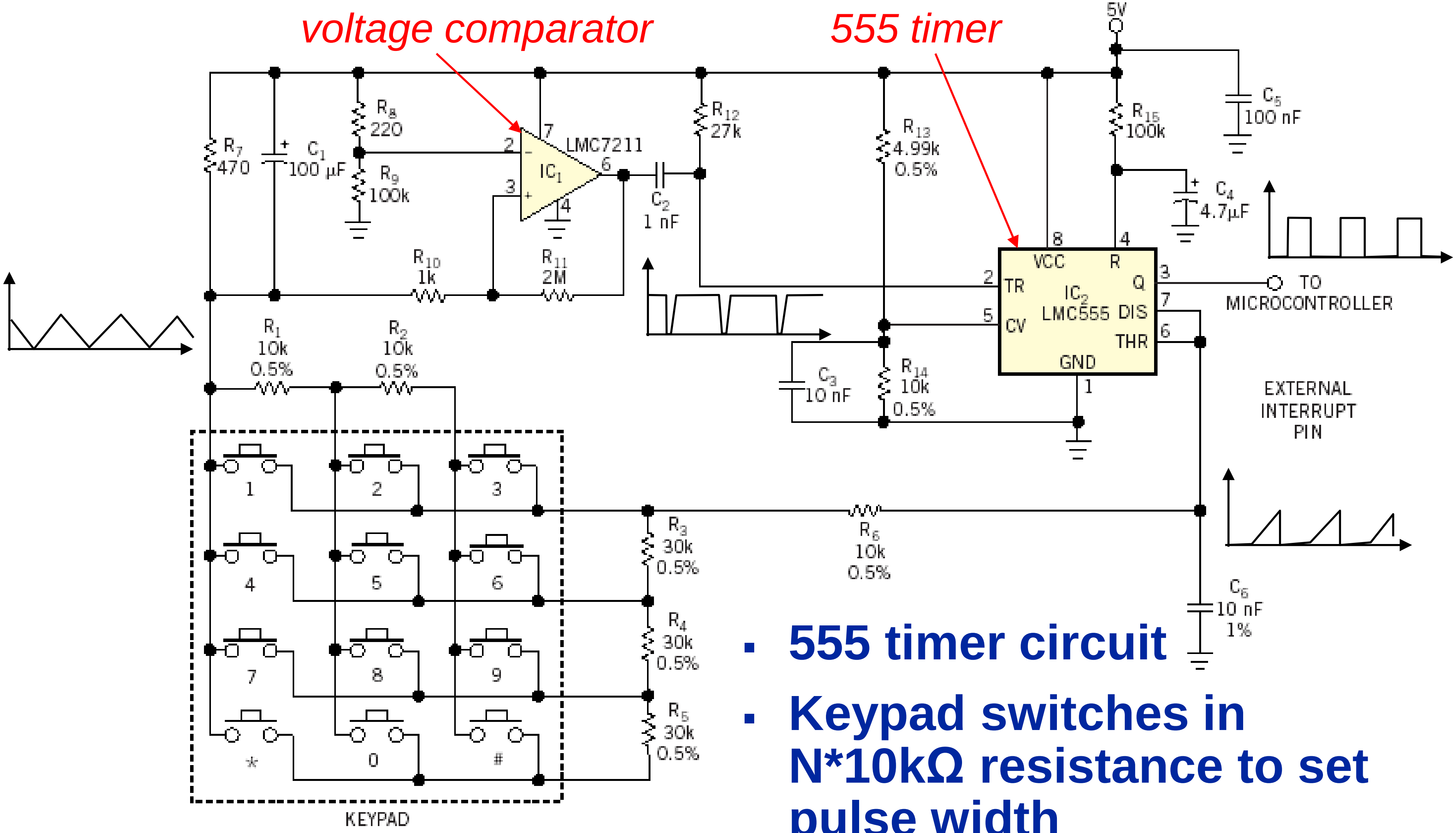
Figures adapted from www.analog.com Application Note 660

Arduino: Pin Change Interrupt Example

```
// Program that counts the number of state changes on an input pin using Pin
// Change Interrupt
#include <PinChangeInt.h>
#include <PinChangeIntConfig.h>

#define inputPin 15 // The pin we are interested in
int stateChanges;

void setup() {
         pinMode(inputPin, INPUT); // Set inputPin to be digital input
         digitalWrite(inputPin, HIGH); // Enable 20KΩ internal pull-up resistor

         // Now we want to attach the interrupt for a change on the pin
         PCintPort::attachInterrupt(inputPin, inputChanged, CHANGE);
         // See: http://playground.arduino.cc/Main/PinChangeInt
         //      https://code.google.com/p/arduino-pinchangeint/wiki/Usage

         stateChanges = 0;   // No state changes yet
         Serial.begin(9600); // Opens serial port, sets data rate to 9600 Baud
}

void inputChanged() {          // Interrupt Service Routine (ISR)...keep it simple!
         stateChanges = stateChanges + 1;    // Increment the # of state changes
         Serial.println(stateChanges);       // Print the # of state changes
}

void loop() {
         coolStuff(); // Main things your program does
}
```

Changes compared to external interrupt example

Single-Wire Keyboard Interface



- **555 timer circuit**
- **Keypad switches in N*10kΩ resistance to set pulse width**

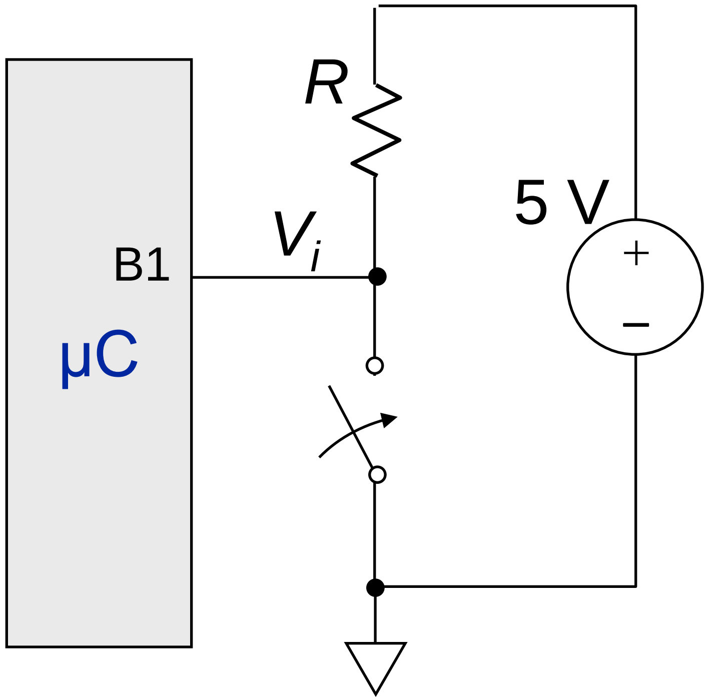
http://www.edn.com/article/CA512131.html, I. Schleicher, EDN 3/31/05

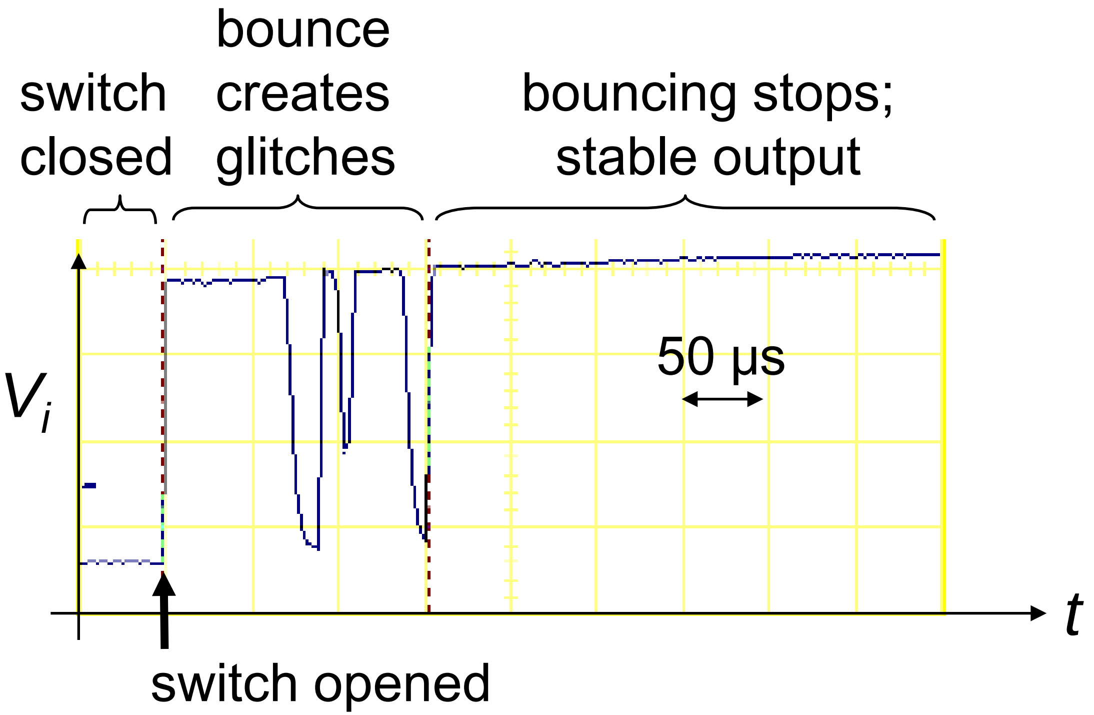
Outline

- Microcontroller Overview
- Event-Based Programming
    - State machines
    - Polling & Interrupts
- Microcontroller Digital I/O
    - On/off I/O
        - LEDs, solenoids, switches (debouncing)
    - Timers
    - Pulse-based I/O
        - PWM motor driving
        - Encoder position feedback
        - Serial and interprocessor communications

Switch Bounce

- **Problem: Signal from switch is not really “digital”**
- **Mechanical switches “bounce”**
  - **Switch temporarily becomes open-circuit**
  - **~Millisecond bouncing**
- **Program may think switch was pressed multiple times**




switch closed

bounce creates glitches

bouncing stops; stable output

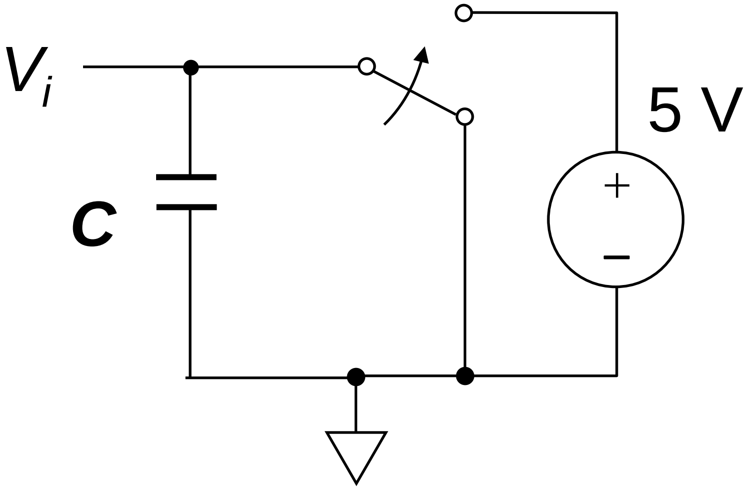
50 µs

switch opened

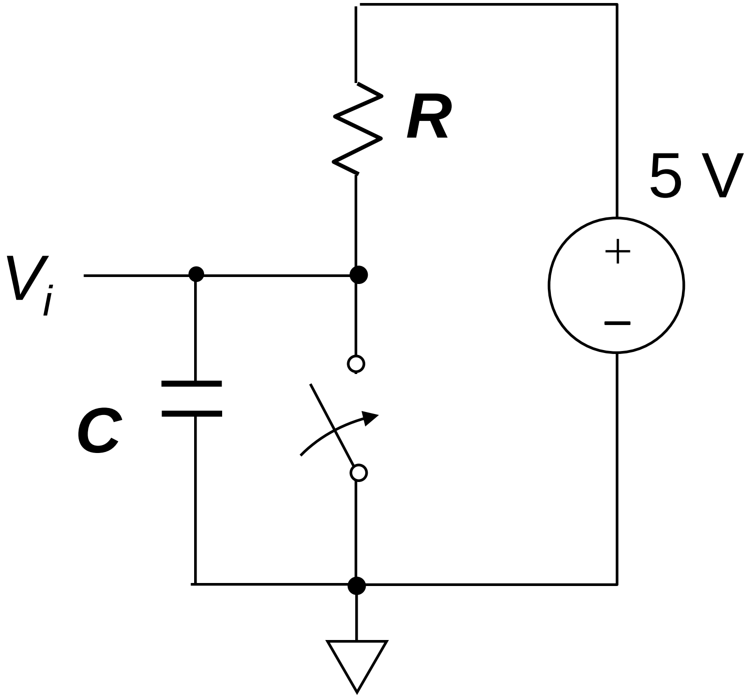
Debouncing in Hardware

- **Add capacitor to hold voltage**
  - **Need single-pole double-throw (SPDT) switch**
- **Pull-up for SPST switch affects speed**
  - **RC around milliseconds leads to slow switching**

Should work for
$C \sim 1\ nF$

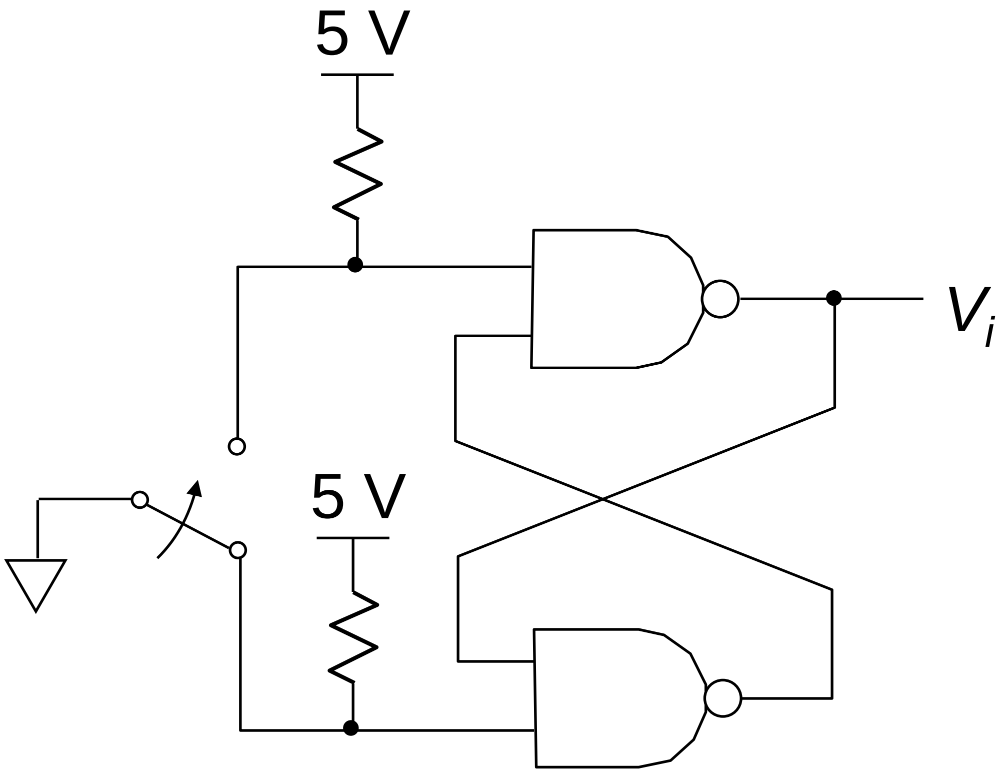


Not recommended



Classical Switch Debounce

- Cross-coupled NAND cell (74[LS,HC,…]00)
    - Also try Schmitt Trigger NAND (74HC132)
- Opened switch will not affect output of cell

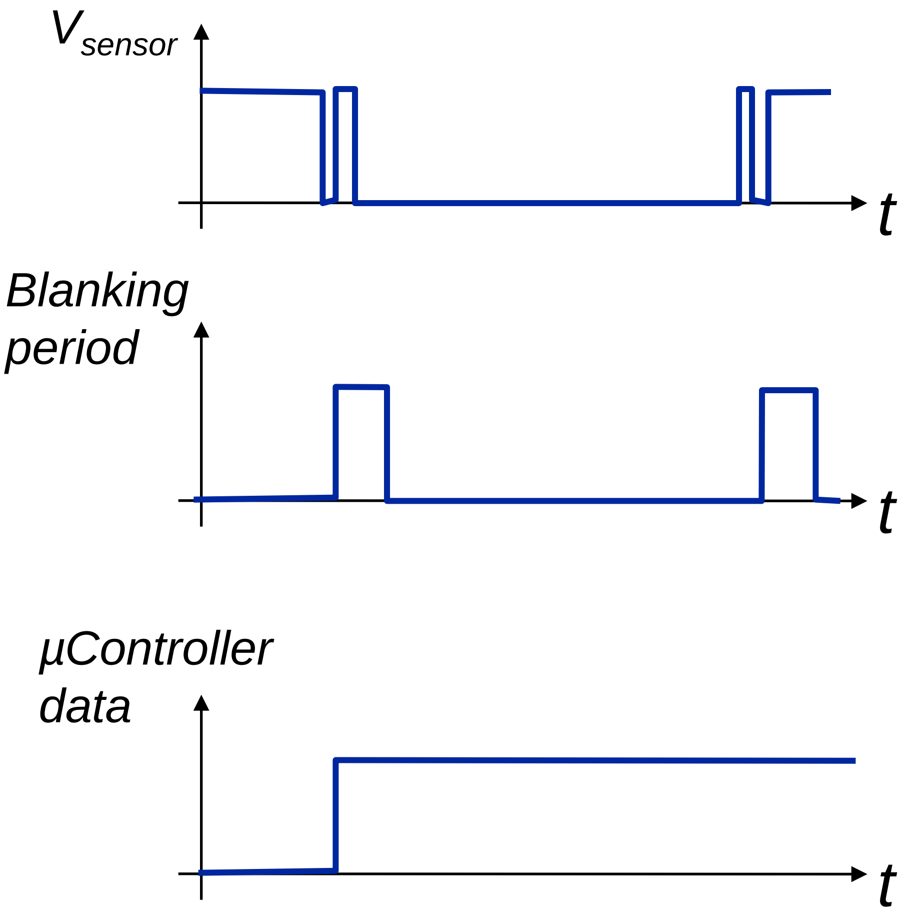


Software Debounce Solution #1

- **Use software delay after detecting first edge**
- **Use in-line delay**
- **Rejects any transitions for this “blanking period”**

- **Drawbacks**
  - **Limits input data speed**
  - **Microcontroller idle during blanking period**

Falling edge input capture example



Arduino Example: Software Debounce #1 (Polling)

```
#define switchPin 3

boolean oldValue;

void setup()
{
  pinMode(switchPin, INPUT);            // Set switchPin to be digital input
  digitalWrite(switchPin, HIGH);        // Enable pull-up resistor
  oldValue = digitalRead(switchPin);    // Read switchPin
  Serial.begin(9600);                   // Open serial port, set data rate
}

void loop()
{
   boolean newValue = digitalRead(switchPin); // Read switchPin
   if (newValue != oldValue)
   {
        Serial.println(“I see a change!”);
        delay(40); // Waits 40 ms for switch to finish bouncing
        oldValue = newValue;
   }
}
```

Arduino Example: Software Debounce #1 (Interrupt)

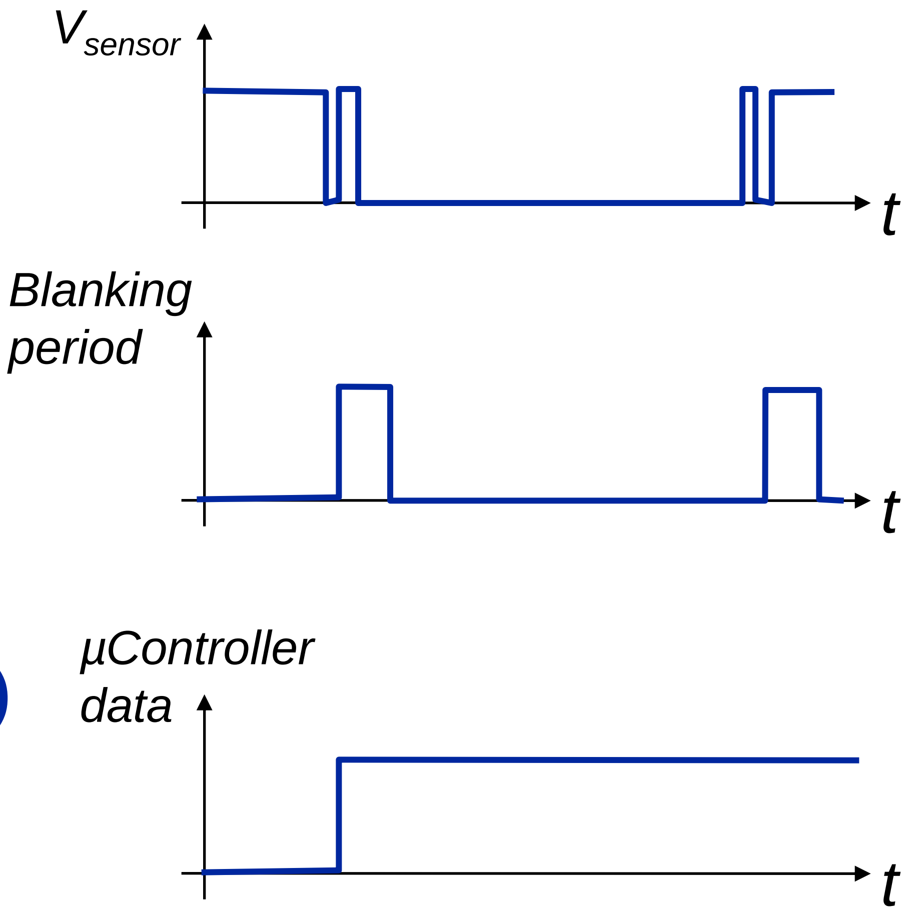
```
#define switchPin 3                  // Corresponds to external interrupt 1

void setup()
{
   pinMode(switchPin, INPUT);         // Set switchPin to be digital input
   digitalWrite(switchPin, HIGH);     // Enable pull-up resistor
   attachInterrupt(1, myISR, CHANGE); // Attach external interrupt
   // See: http://www.arduino.cc/en/Reference/AttachInterrupt
   Serial.begin(9600);                // Open serial port, set data rate
}

void myISR()
{
   Serial.println(“I see a change!”);
   delayMicroseconds(40000); // Waits 40 ms for switch to finish bouncing
}

void loop()
{ // Free to do other stuff here }
```

Software Debounce Solution #2

- Disable edge interrupt after first edge
- Set and start timer
- Use timer interrupt to define blanking period
- Clear and re-enable edge interrupts after blanking period

- More elegant solution
- No delay inside ISR or loop()
- Drawbacks:
  - More complicated
  - Occupies a timer

Falling edge input capture example

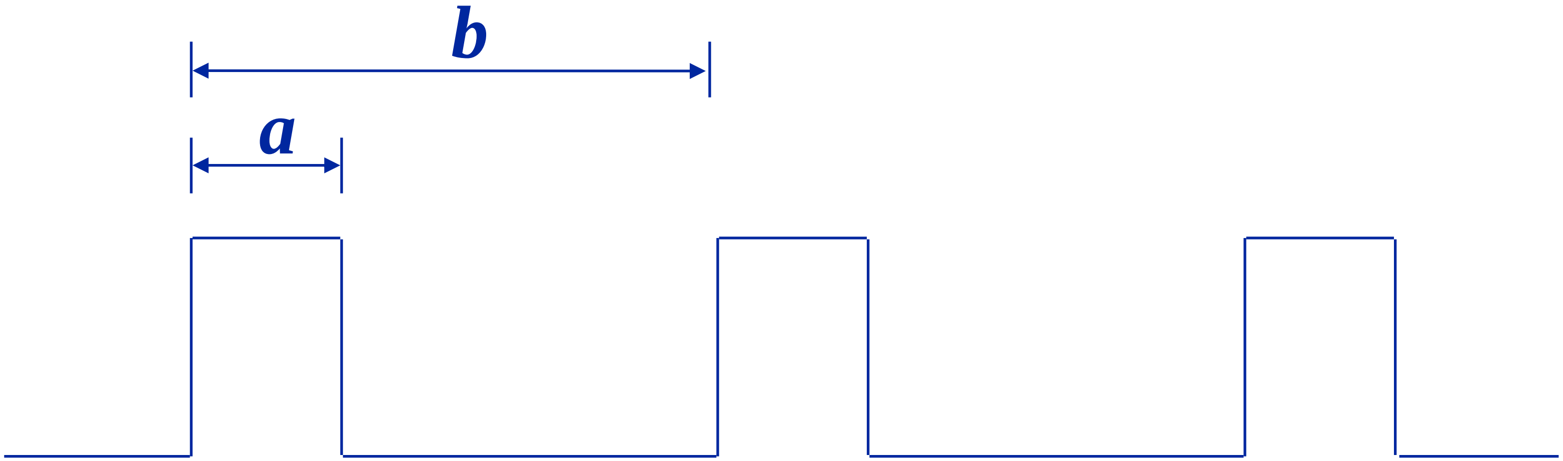


Outline

- Microcontroller Overview
- Event-Based Programming
  - State machines
  - Polling & Interrupts
- Microcontroller Digital I/O
  - On/off I/O
    - LEDs, solenoids, switches (debouncing)
  - Timers
  - Pulse-based I/O
    - PWM motor driving
    - Encoder position feedback
    - Serial and interprocessor communications

Pulse Width Modulation (PWM)

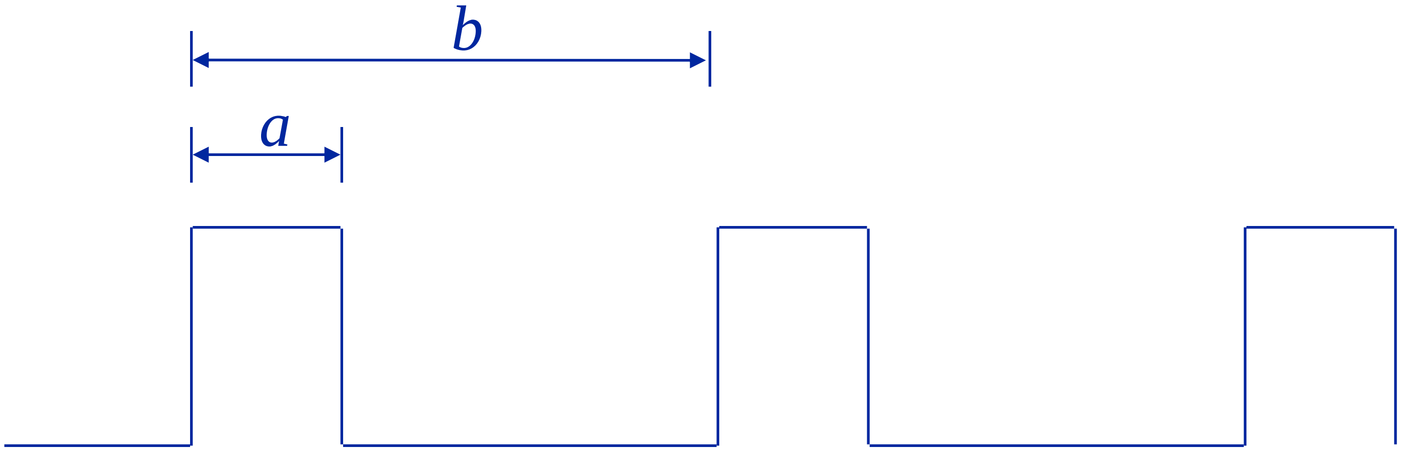
- Commonly used as a control signal for DC motors
- Microcontrollers often have built-in PWM modules



$$\text{Duty cycle} = \frac{a}{b} \times 100\%$$

$$\text{Frequency} = \frac{1}{b} \quad \text{(usually held constant)}$$

PIC18: PWM Generator

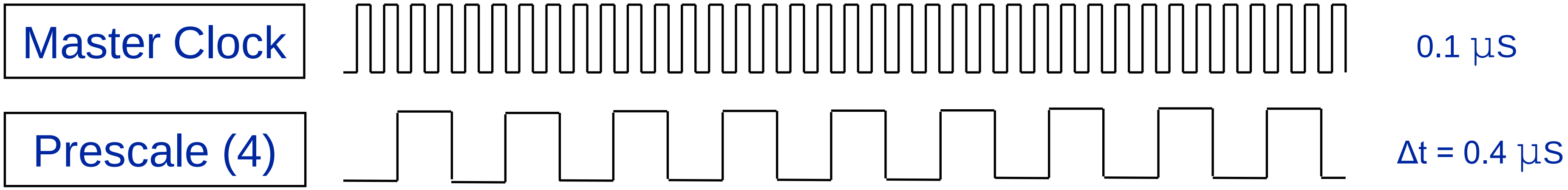


$$\text{Duty cycle} = \frac{a}{b} \times 100\% \qquad \text{Freq} = \frac{1}{b}$$

- Configure Capture-Compare module for PWM
  - `setup_ccp1(CCP_PWM)`
- Set up PWM period, b:
  - `setup_timer_2(prescale, period, postscale)`
- Set up PWM duty cycle, a:
  - `set_pwm1_duty(duty_high)`

PIC18: PWM Example Calculation

- Assume Fosc = 40MHz with #fuses HS compiler directive
    - Master Clock = 10MHz, Period = 0.1 μsec
- setup_timer_2(T2_DIV_BY_4,124,postscale)
    - Really just a series of clock dividers

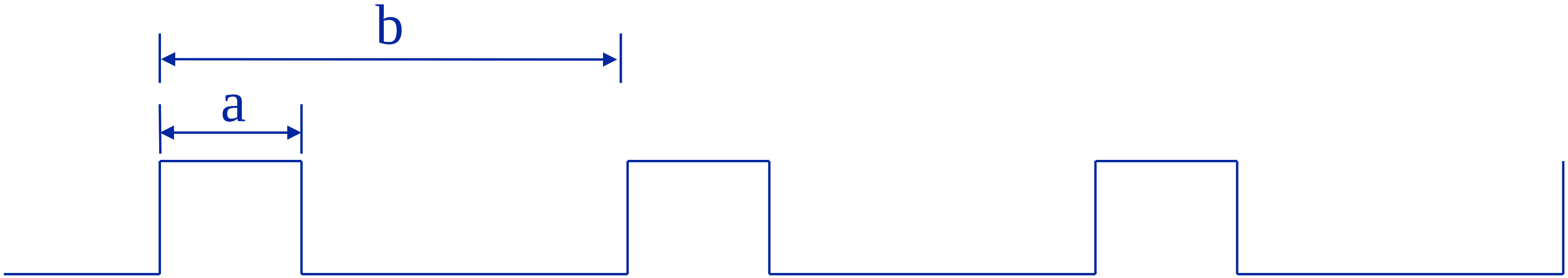

Master Clock

Prescale (4)

0.1 μS

Δt = 0.4 μS

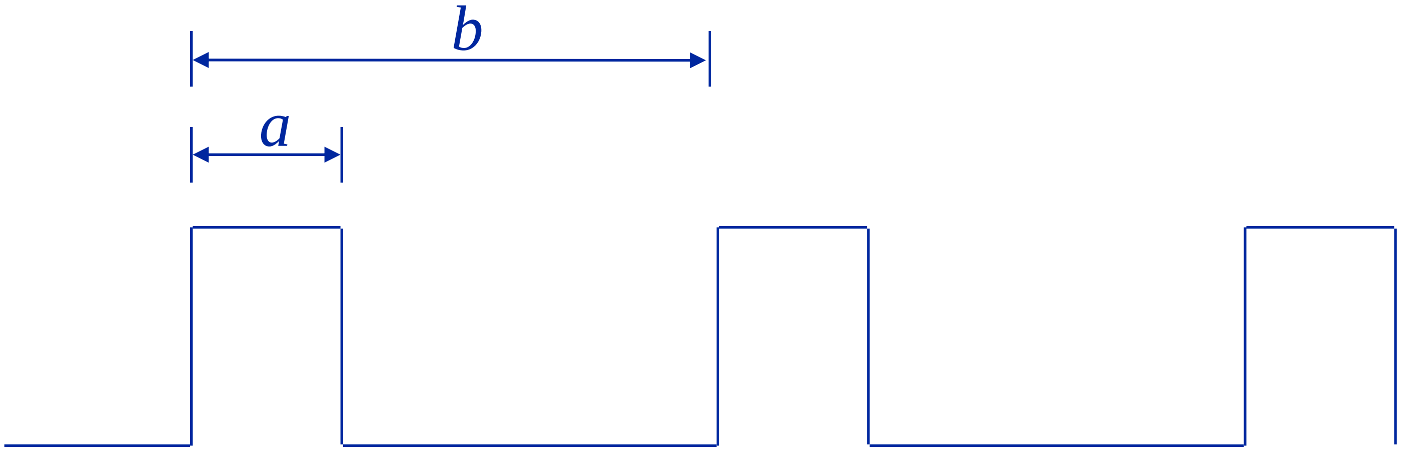
- Period is in units of Δt as determined by Prescale
    - period = 124 → b = (124+1)(0.4μs) = 50 μs → 20kHz
- For PWM, duty cycle is also in units of Δt
    - set_pwm1_duty(30) → a = (30)(0.4μs) = 12 μs = 24%



PIC18: PWM Example Code

```
#include <18F4431.H>
#fuses  HS,NOWDT,NOPROTECT,NOLVP
#use delay(clock=40000000)
#use rs232(baud=9600, xmit=PIN_C6, rcv=PIN_C7, INVERT)

void main() {
 disable_interrupts(global);

 // Set the Compare/Capture/PWM Module to PWM Mode
 setup_ccp1(CCP_PWM);

 // setup_timer_2 (prescale, period register value, postscale value)
 // Master clock cycle = (4 / 40MHz) = 100 ns
 // Timer clock cycle  = (prescale) * (Master Cycle) = 4 * 100 ns = 0.4 µs
 // Timer reset period = (period reg val+1) * (Timer Cycle) = (124+1) * 0.4 µs = 50 µs
 // Timer interrupt period = (postscale) * (Timer Period) = 16 * 50 µs = 0.8 ms (unused)
 setup_timer_2(T2_DIV_BY_4,124,16);

 enable_interrupts(GLOBAL);

 // Duty cycle = (arg of set_pwm1_duty) * (Timer Cycle) = 30 * 0.4 µs = 12 µs
 // ➔ Duty cycle of 24% (pwm1_duty/timer reset period = 12 µs/50 µs)
 set_pwm1_duty (30);

 // PWM signal should now be on Pin 17 (C2/CCP1)

 // Don’t forget to hang out in an infinite loop, otherwise the µC will halt
 while(TRUE)
   delay_ms(1000);
}
```

Arduino: PWM Generator



$$\text{Duty cycle} = \frac{a}{b} \times 100\% \qquad \text{Freq} = \frac{1}{b}$$

- **All (simple cases) handled by the built-in libraries**
  - **DC motors: analogWrite(pin, Value) function**
    - **Value ranges from 0 (0% duty cycle) to 255 (100%)**
    - **Frequency (fixed) = ~500Hz ➔ Period = ~2ms**
  - **RC servos: Servo library**
- **For other cases, see the datasheet**

Arduino: PWM for Variable-Brightness LED

```
int LEDPin = 9;      // LED connected to digital pin 9
int analogPin = 3;   // Potentiometer connected to analog pin 3
int val = 0;         // Variable to store the read value

void setup()
{
  pinMode(LEDPin, OUTPUT);   // Sets the pin as output
}

void loop()
{
  val = analogRead(analogPin);   // Read the potentiometer setting
  analogWrite(LEDPin, val / 4);
        // analogRead values go from 0 to 1023 (10 bits)
        // analogWrite values go from 0 to 255 (8 bits)
} //http://www.arduino.cc/en/Reference/AnalogWrite
```

Arduino: Timer1 Library

- Can find it here: http://playground.arduino.cc/code/timer1
- Most important routines:
  - initialize(period): enable use of the below functions
  - setPeriod(period): set the period
  - pwm(pin, duty, period): generates PWM on spec'd pin
  - setPwmDuty(pin, duty): shortcut for setting duty
  - attachInterrupt(function, period): calls function at interval specified by period
  - detachInterrupt(): disables the attached interrupt
  - disablePwm(pin): turn PWM off on that pin
- Notes
  - The period ('period') is b in microseconds
  - The duty cycle ('duty') is a 10-bit value (0 to 1023) (a = (duty/1023) * b
  - Breaks analogWrite() for digital pins 9 & 10

Arduino: PWM using Timer1 Library

```
#include “TimerOne.h”

int LEDPin = 9;      // LED connected to digital pin 9
int analogPin = 3;   // Potentiometer connected to analog pin 3
int val = 0;         // Variable to store the read value

void setup()
{
  Timer1.initialize(200);  // Init. Timer1 and set a 0.2-msec period
  Timer1.pwm(LEDPin, 0);   // Set up PWM on LED pin w/ 0% duty cycle
}

void loop()
{
  val = analogRead(analogPin);   // Read the potentiometer setting
  Timer1.setPwmDuty(LEDPin, val);
        // analogRead values go from 0 to 1023 (10 bits)
        // setPwmDuty values go from 0 to 1023 (10 bits)
}
```

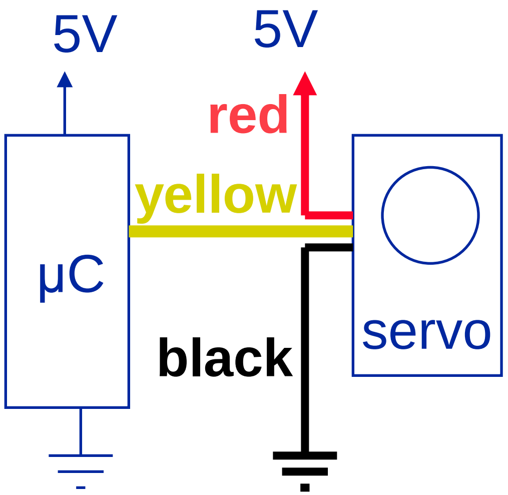
Driving an R/C Servo

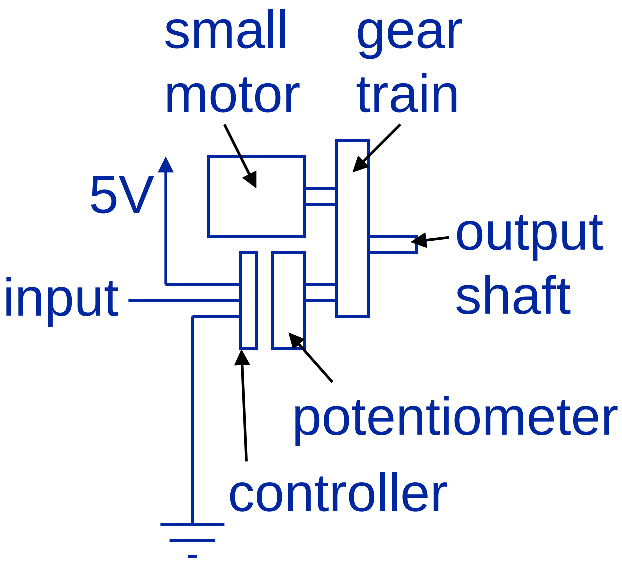
- Pulse width, t_{pw}, mapped to angle:

  1.00 ms ↔ 0°

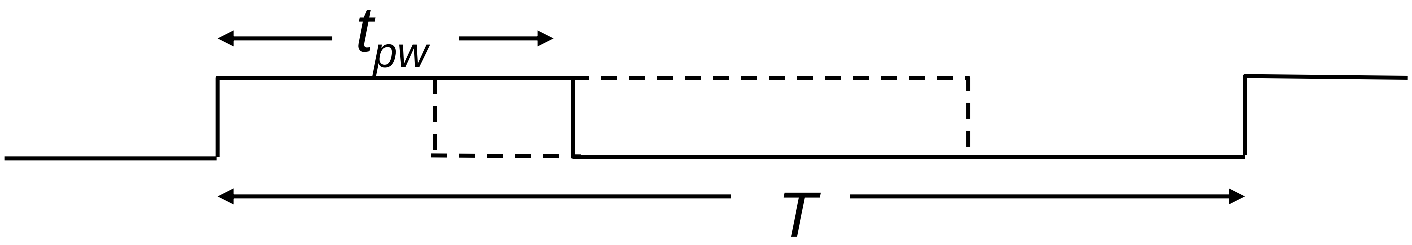
  1.50 ms ↔ 90°

  2.00 ms ↔ 180°





- Pulse refresh period *T* nominally 20 ms, and needs to be within 4 ms < *T* < 30 ms range or servo may jitter

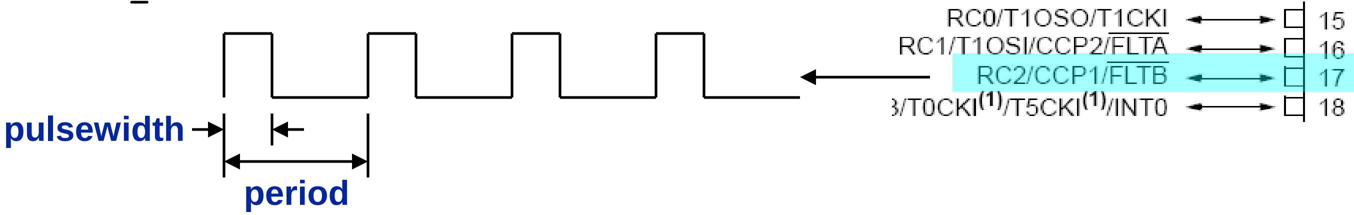


Arduino: RC Servo Example

```
#include <Servo.h>

Servo myServo;

void setup() {
        myServo.attach(9);    // “Attach” the servo to Pin 9
        myServo.write(90);    // Set servo to midpoint (argument is degrees)
        Serial.begin(9600);   // Open serial port and set data rate
} // See http://www.arduino.cc/playground/ComponentLib/servo

void loop() {
        Serial.println("Starting!");
        while (true) {
                Serial.println("Move to 0 degrees");
                myServo.write(0);
                delay(2000);  // Wait 2 seconds

                Serial.println("Move to 90 degrees");
                myServo.write(90);
                delay(2000);  // Wait 2 seconds

                Serial.println("Move to 180 degrees");
                myServo.write(180);
                delay(2000);  // Wait 2 seconds
        }
}
```

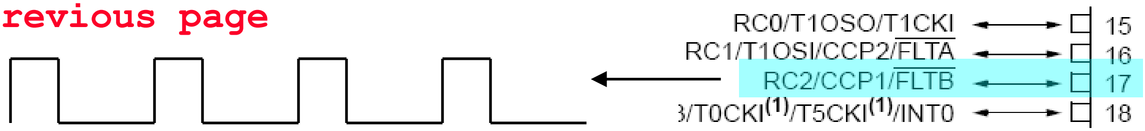
PIC18: RC Servo Example (p. 1/2)

```
#include <18F4431.h>
#fuses XT,NOWDT,NOPROTECT,NOLVP // note that your crystal and delay may be different
#use delay(clock=4000000)
#use rs232(baud=9600, xmit=PIN_C6, rcv=PIN_C7)

#define SERVO_0   1000
#define SERVO_90  1500
#define SERVO_180 2000

int16 pulse_width;
int16 period = 20000; // 20 milliseconds with 4 MHz clock
int1 state = 0;

#INT_CCP1
void pulse_ISR() {     // Flips the state of the PWM pin
         if (state == 0) {
                  setup_CCP1(CCP_COMPARE_SET_ON_MATCH);
                  CCP_1 = 0;  set_timer1(0); // Set CCP1 high
                  setup_CCP1(CCP_COMPARE_CLR_ON_MATCH);
                  CCP_1 = pulse_width; // Set how long CCP1 will be high
                  state = 1;
         }
         else if (state == 1) {
                  setup_CCP1(CCP_COMPARE_CLR_ON_MATCH);
                  CCP_1 = 0;  set_timer1(0); // Set CCP1 low
                  setup_CCP1(CCP_COMPARE_SET_ON_MATCH);
                  CCP_1 = period - pulse_width; // Set how long CCP1 will be low
                  state = 0;
         }
}
```



PIC18: RC Servo Example (p. 2/2)

```
#INT_CCP1
void pulse_ISR() {...      // see previous page
         }

void main() {
         set_tris_c(0b10000100); // Set CCP1 as output
         setup_timer_1(T1_INTERNAL);
         enable_interrupts(GLOBAL);
         enable_interrupts(INT_CCP1);
         setup_CCP1(CCP_COMPARE_CLR_ON_MATCH);
         CCP_1 = 0; set_timer1(0);   // Cause match, clear CCP1, generate CCP interrupt

         printf("Starting!\r\n");
         while (TRUE) {
                  printf("Move to 0 degrees\r\n");
                  pulse_width = SERVO_0;
                  delay_ms(2000);

                  printf("Move to 90 degrees\r\n");
                  pulse_width = SERVO_90;
                  delay_ms(2000);

                  printf("Move to 180 degrees\r\n");
                  pulse_width = SERVO_180;
                  delay_ms(2000);
         }
}
```

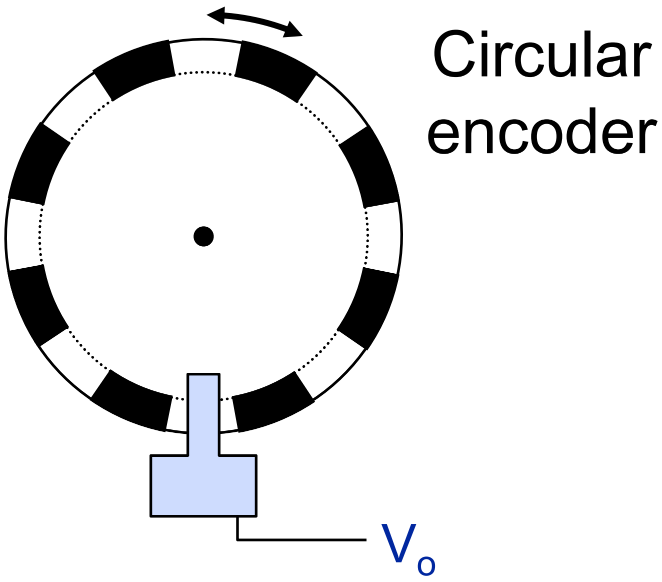


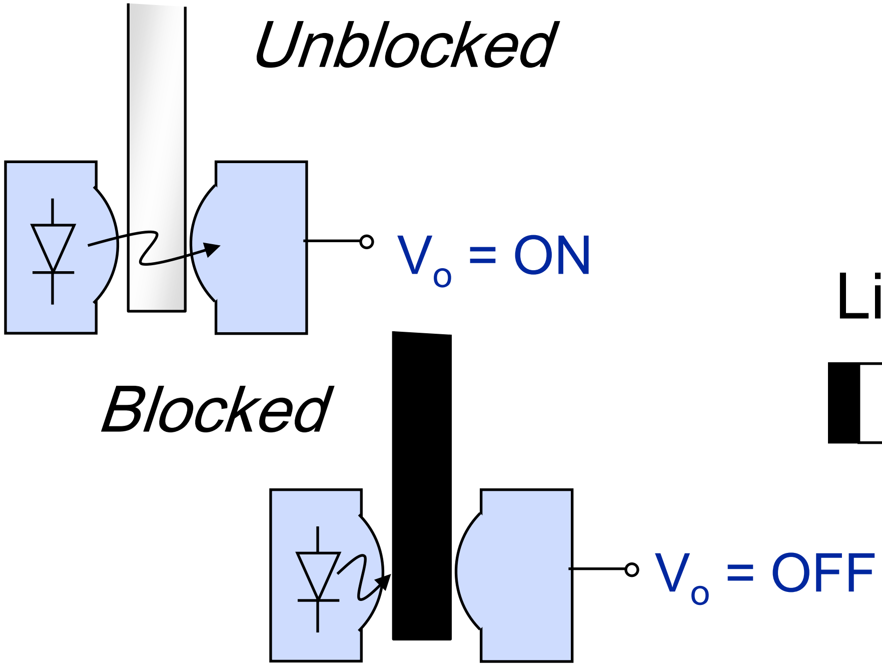
Outline

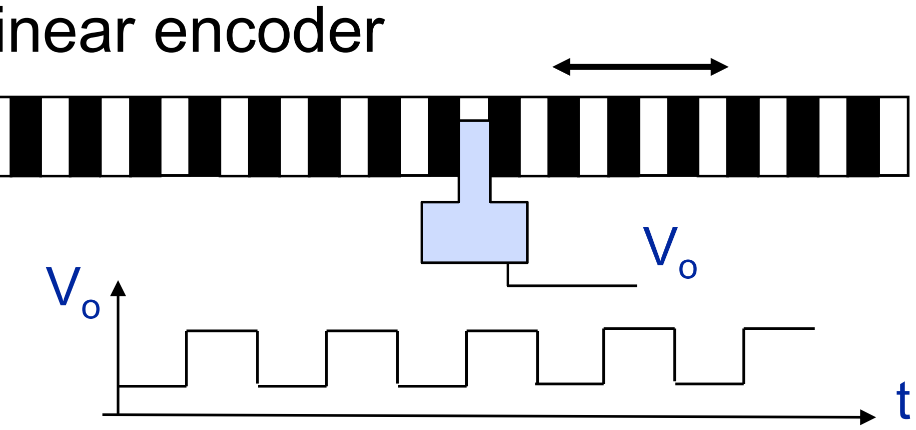
- Microcontroller Overview
- Event-Based Programming
  - State machines
  - Polling & Interrupts
- Microcontroller Digital I/O
  - On/off I/O
    - LEDs, solenoids, switches (debouncing)
  - Timers
  - Pulse-based I/O
    - PWM motor driving
    - Encoder position feedback
    - Serial and interprocessor communications

Encoders

- **Infrared emitter and detector placed across gap**
- **Moving pattern between gap modulates light**
- **Motion is detected by electronically counting pulses of light**

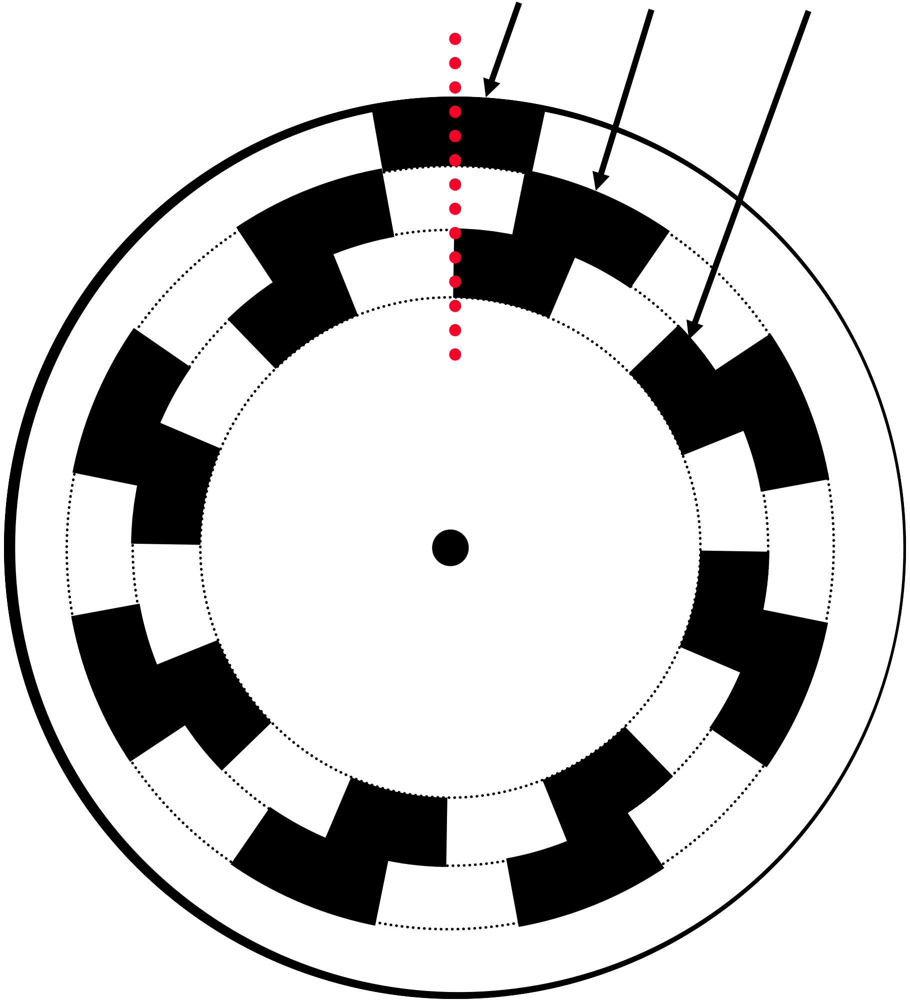




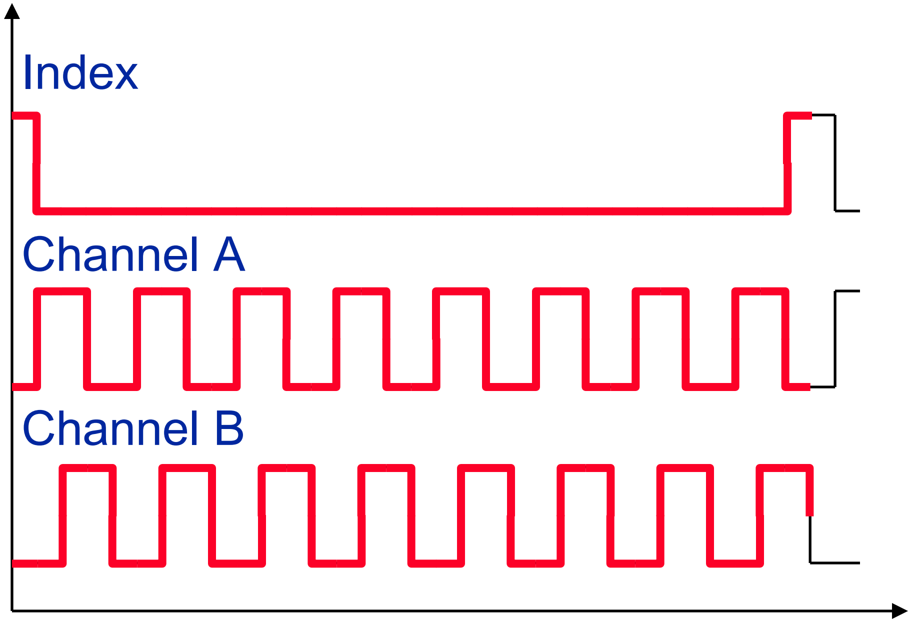


Circular Encoder Operation

- Three channels: Index, A and B



- Dark is logic high (circuit - dependent)
- Frequency gives speed



Clockwise

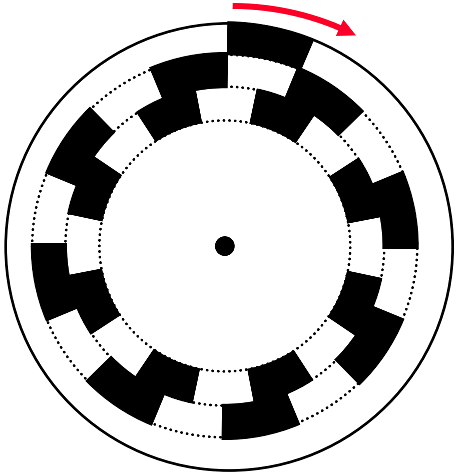
Phase	A	B
1	1	0
2	1	1
3	0	1
4	0	0

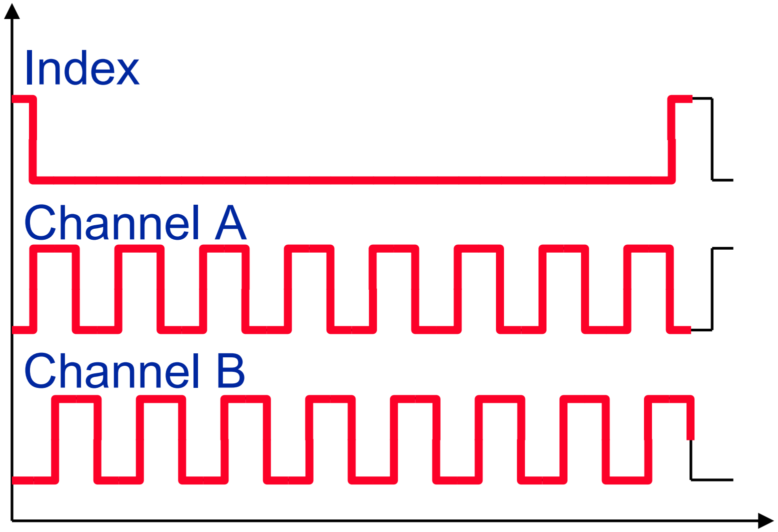
Counter-clockwise

Phase	A	B
1	0	0
2	0	1
3	1	1
4	1	0

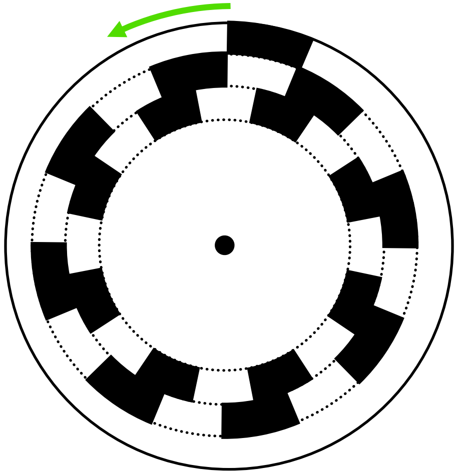
Quadrature Encoder

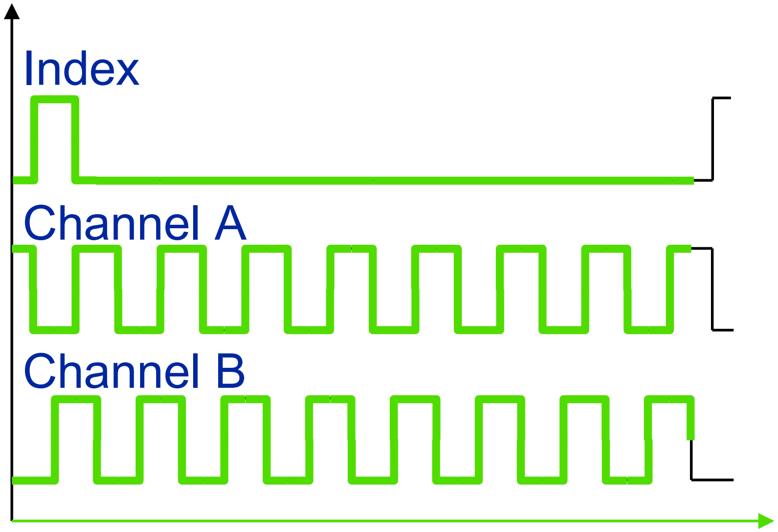
- A and B are in quadrature (90° phase difference) to detect direction
- Clockwise: A leads B





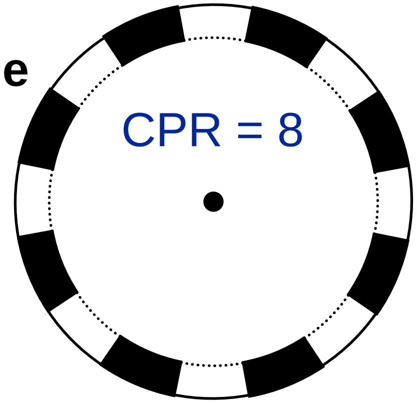
- Counterclockwise: B leads A





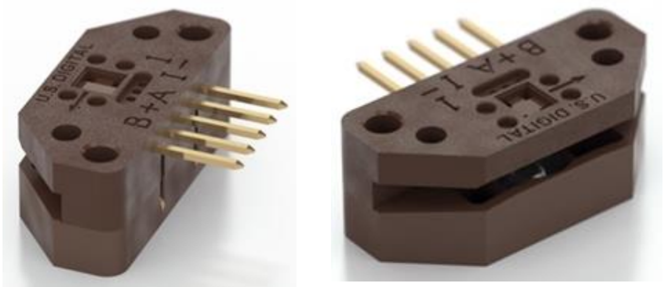
Encoder Terminology

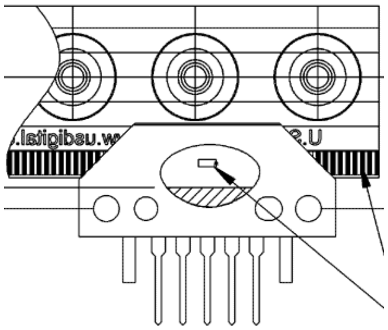
- CPR: Counts per revolution, i.e., the number of cycles or “pulses” on one encoder track (disc below has CPR=8, not 16)
- Resolution
    - Referring to a mechanical encoder disk (whether single-track or double-track [i.e., quadrature]) → same as CPR
    - Referring to output resolution after decoding, see below
- x1, x2, or x4 (i.e., quadrature) decoding
    - Depends on decoding software or hardware
    - x1 means counting pulses (rising or falling edges, but not both) on a single track → Resolution = CPR
    - x2 means counting rising and falling edges on a single track → Resolution = 2 x CPR
    - x4 means counting rising and falling edges on both tracks (i.e., quadrature) → Resolution = 4 x CPR

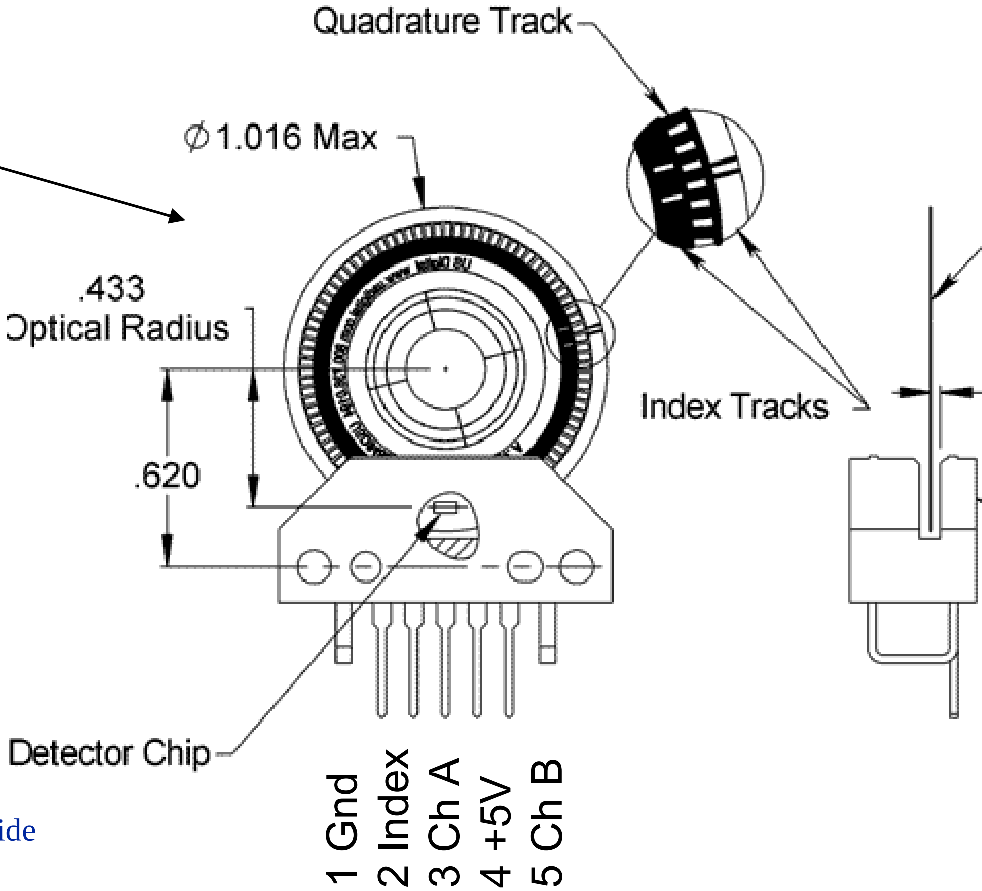


U.S. Digital EM1 Encoders

- Well-known optical encoders
  - e.g., US Digital & many others sell them
- Various counts/revolution
  - up to 1250 for 1” disks, 2500 for 2” disks
  - up to 500 counts/inch for linear
  - $35-40
  - ~$10 for code wheel or code strip





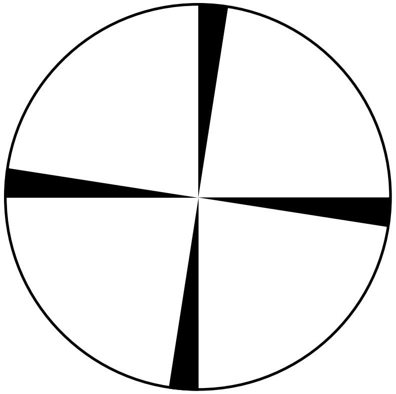


http://usdigital.com/support/resource-library/em1-heds-reference-guide

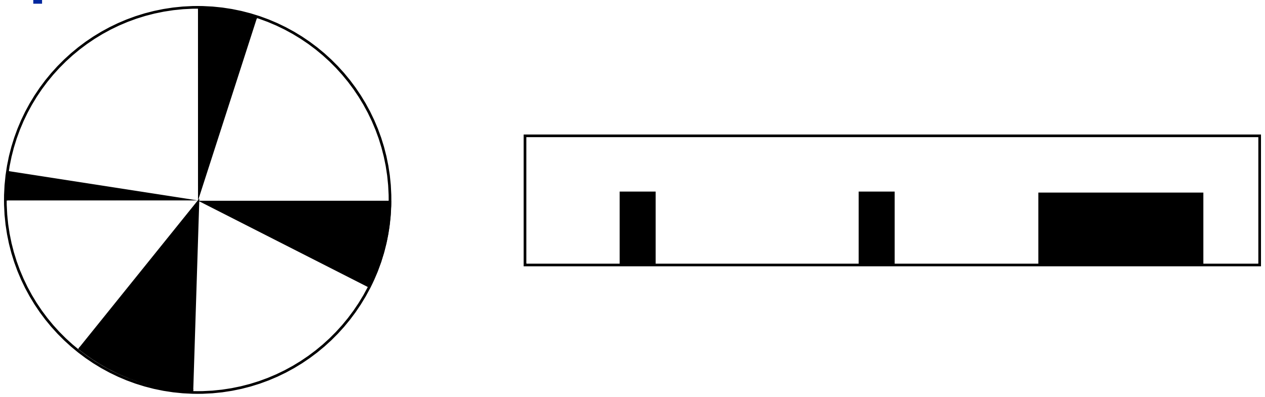
Figures from www.usdigital.com

Custom Encoder Disks and Linear Strips

- Can make custom disks and strips
  - plastic with opaque paint or tape
  - metal with notches
- You can make an encoder with just a few divisions
  - This encoder suffices for four positions



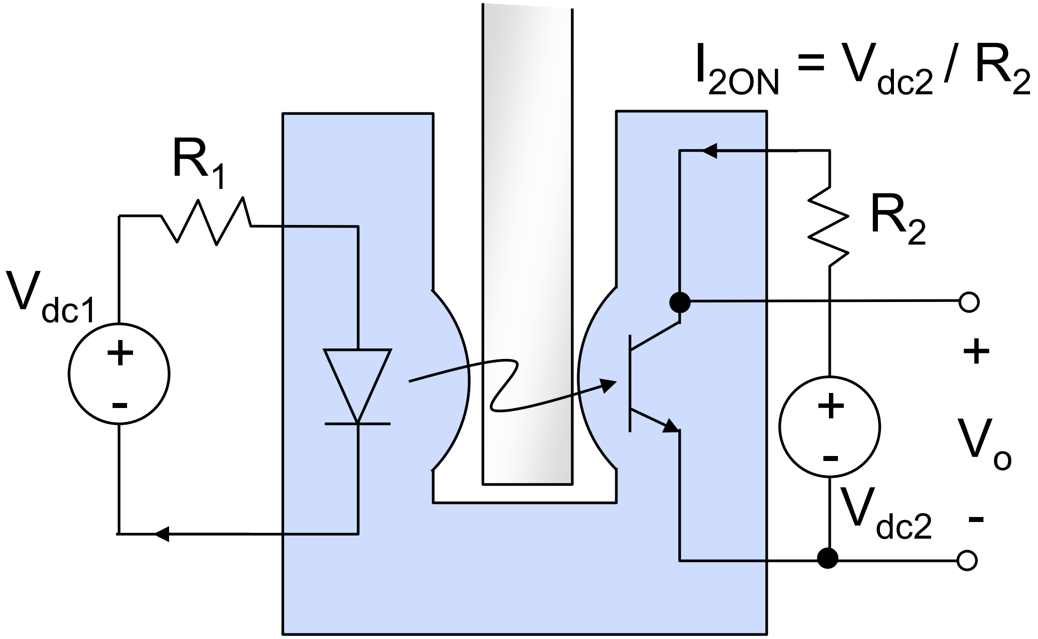
- Could encode positions with different widths…



IR Emitter-Detector (e.g., Omron EE-SX1042)

- Available in individual emitter/detector pairs or in slot packages
- Many components require external resistors to set current
- Set R_1 and R_2 such that current ratings are not exceeded

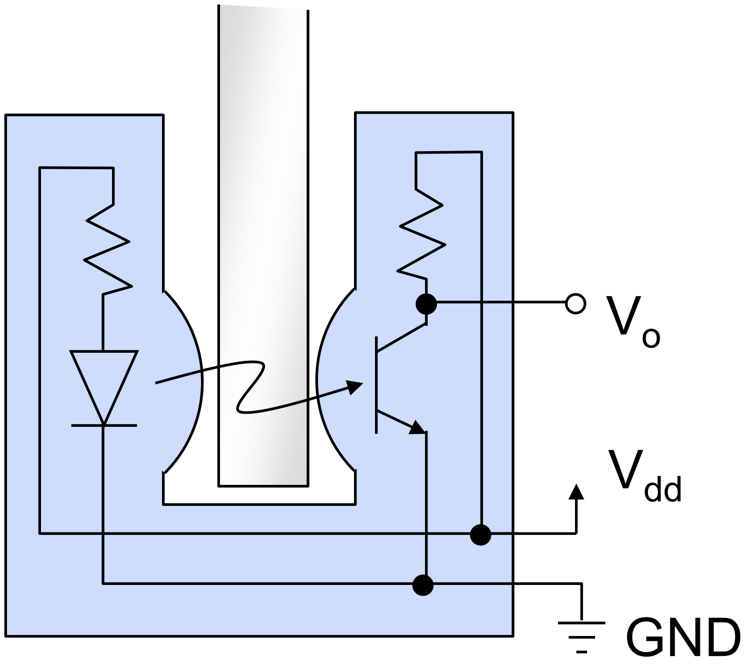
Component type #1



$I_{2ON} = V_{dc2} / R_2$

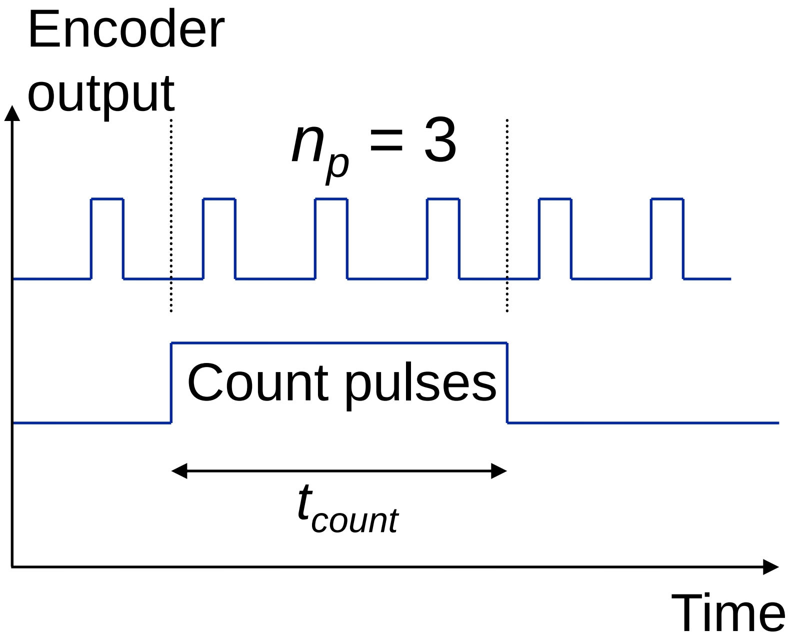
$I_{LED} = (V_{dc1} – V_{LED}) / R_1$

Component type #2



Measuring Shaft Speed – Method 1 (Frequency Counting)

- ***Count the number of encoder pulses during a fixed time interval***
- ***Number of counted pulses is directly proportional to rotational speed***



$$v_m = \frac{60 \cdot n_p}{N \cdot t_{count}}$$

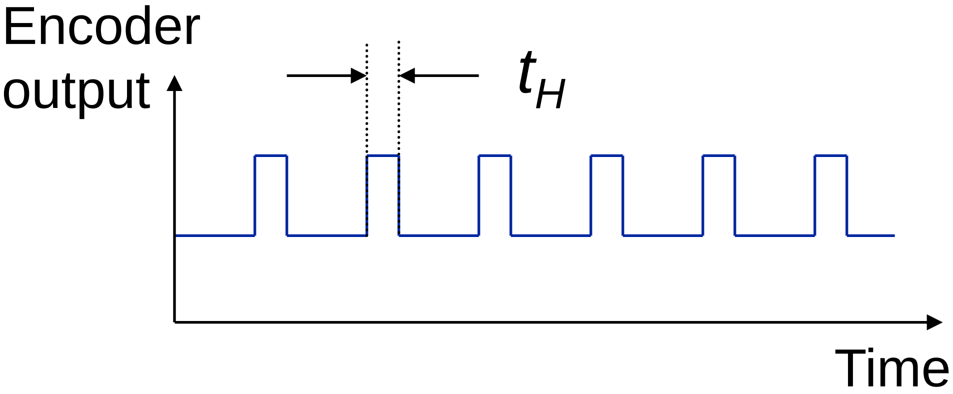
N = encoder counts per rev.
n_p = number of pulses counted
v_m = motor speed in rpm
t_{count} = counting interval, sec.

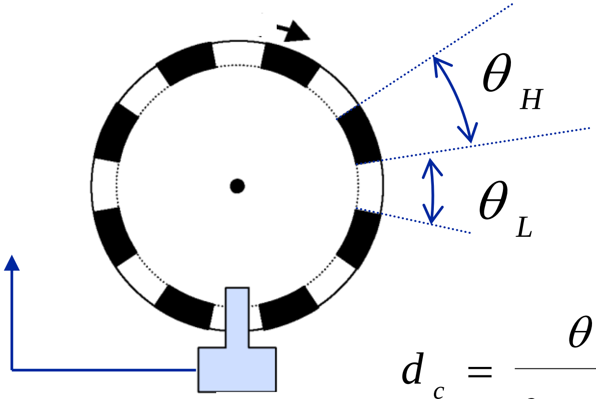
Arduino: Measuring Shaft Speed (Frequency Counting)

```
#define inputPin 2          // Arduino pin for external interrupt 0
int counts_per_rev = 32;    // Varies depending on encoder
int count;

void setup() {
   pinMode(inputPin, INPUT);     // Set inputPin to be digital input
   digitalWrite(inputPin, HIGH); // Enable 20KΩ internal pull-up resistor

   // Now we want to attach the interrupt for a rising edge on the pin
   attachInterrupt(0, countPulses, RISING); // Also avail.: LOW, CHANGE, FALLING
   // See: http://www.arduino.cc/en/Reference/AttachInterrupt
}

void countPulses() {    // ISR
   count = count + 1;   // Increment the # of encoder ticks counted
}

void loop()
{
   count = 0;     // Start a fresh count
   delay(250);   // Wait for ¼ sec. while the ISR counts ticks
   float rpm = (60*count)/(counts_per_rev*0.25);  // Find revolutions per minute
}
```

Measuring Shaft Speed – Method 2 (Pulsewidth-based)

- *Measure the pulse width of the encoder pulses*
- *Pulse width is inversely proportional to rotational speed*





$$d_c = \frac{\theta_H}{\theta_H + \theta_L}$$

N = encoder counts per rev.

d_c = duty cycle of encoder signal

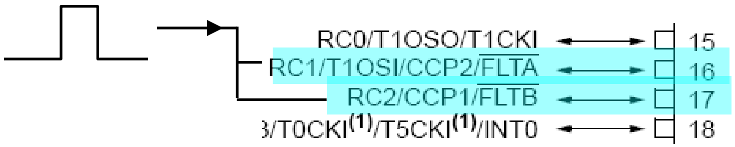
v_m = motor speed in rpm

$$v_m = \frac{60 \cdot d_c}{N \cdot t_H}$$

PIC18: Measuring Shaft Speed (Pulse Width Measurement)

Note that CCP_1 and CCP_2 registers capture regardless of the interrupt

```
#include <18F4431.H>
#fuses  H4,NOWDT,NOPROTECT,NOLVP
#use delay(clock=40000000)
#use rs232(baud=9600, xmit=PIN_C6, rcv=PIN_C7)
long rise,fall,pulse_width;
#int_ccp2
void isr()
{
   rise = CCP_1;                // CCP_1 is the time the pulse went high
   fall = CCP_2;                // CCP_2 is the time the pulse went low
   pulse_width = fall - rise;   // pulse_width/(clock/4) is the time
}
// In order for this to work the ISR overhead must be less than the
// low time.  For this program the overhead is 45 instructions.
void main() {
   printf("\n\rHigh time (sampled every second):\n\r");
   setup_ccp1(CCP_CAPTURE_RE);    // Configure CCP1 to capture rise
   setup_ccp2(CCP_CAPTURE_FE);    // Configure CCP2 to capture fall
   setup_timer_1(T1_INTERNAL);    // Start timer 1

   enable_interrupts(INT_CCP2);   // Enable CCP2 (falling edge) interrupt
   enable_interrupts(GLOBAL);

   while(TRUE) {
      delay_ms(1000);
      printf("\r%lu us ", pulse_width );
   }
}
```



Arduino: Measuring Shaft Speed (Pulse Width Measurement)

```
int pin = 7;                // Input pin
int ticksPerRev = 32;       // Varies depending on encoder
int dc = 0.5;               // Encoder “duty cycle”
unsigned long duration; // Pulse duration in microseconds

void setup()
{
       pinMode(pin, INPUT);    // Sets pin 7 to be an input
}

void loop()
{
       duration = pulseIn(pin, HIGH);  // Microseconds per pulsewidth
       //(See http://arduino.cc/en/Reference/pulseIn)

       // Rotations per minute
       float frequency = (60*1000000* dc)/ (duration * ticksPerRev);
}
```

Note:

- pulseIn() is blocking!
- Interrupts would prevent delays in loop()

Summary

- Event-Based Programming
  - State machines
  - Polling & Interrupts
- Microcontroller Digital I/O
  - On/off I/O (LEDs, solenoids, switches)
  - Pulse-based I/O (PWM, encoders, serial)
- Relevant tools: timer/counters, output compare, input capture, interrupts

Announcements

- Microcontroller Familiarization Task
  - TAs will be distributing kits
  - Refer to Canvas for details
  - All states must be working as specified to receive full credit