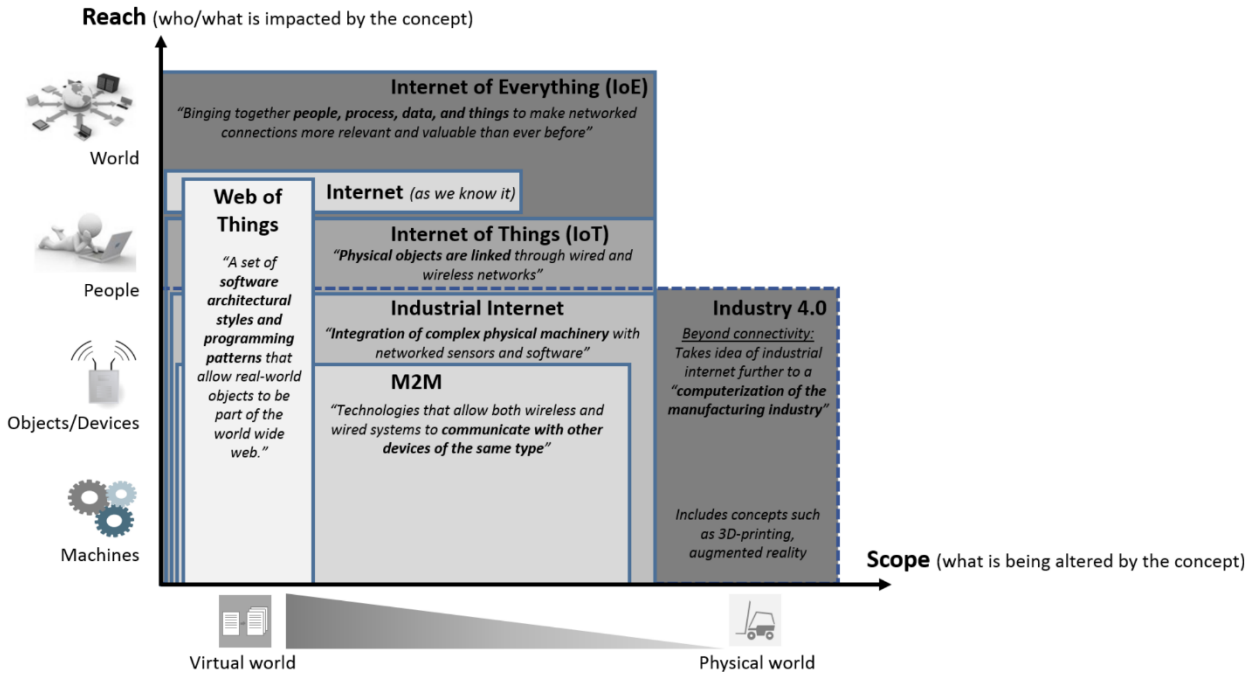
IoT Devices and Applications

IoT Definition

The Internet of Things (IoT) describes the network of physical objects that are embedded with sensors, software, and other technologies for the purpose of connecting and exchanging data with other devices and systems over the Internet.



IoT Applications

Energy
- Smart meters
- Distribution automation
- EV charging Mgt
- Solar power
- Battery Storage Mgt

Automotive
- Navigation
- Tolling
- Traffic management
- Parking systems
- Drive assistance systems
- OBD and assistance

Cities & Communities
- Public transport
- Smart buildings
- Smart government
- Utility and power supply
- Healthcare and emergency assistance
- Public access services

Healthcare
- Implants
- Vitals monitoring
- Telemedicine
- Remote diagnostics
- Equipment tracking
- Equipment status monitoring

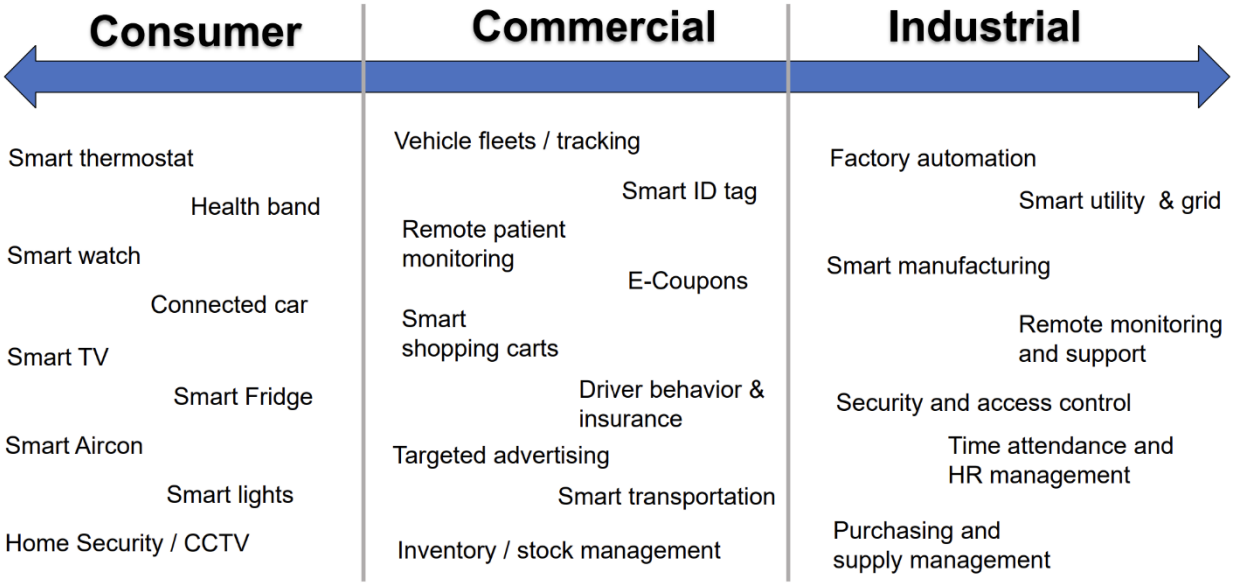
Industrial
- Asset utilization
- Inventory and stock management
- Just-in-time manufacturing
- Location aware safety
- Smart tags & production monitoring
- Smart pumps / valves etc.
- Energy management

Security
- Surveillance & tracking
- Access control
- Emergency services
- Environmental Monitoring
- Disaster management and response

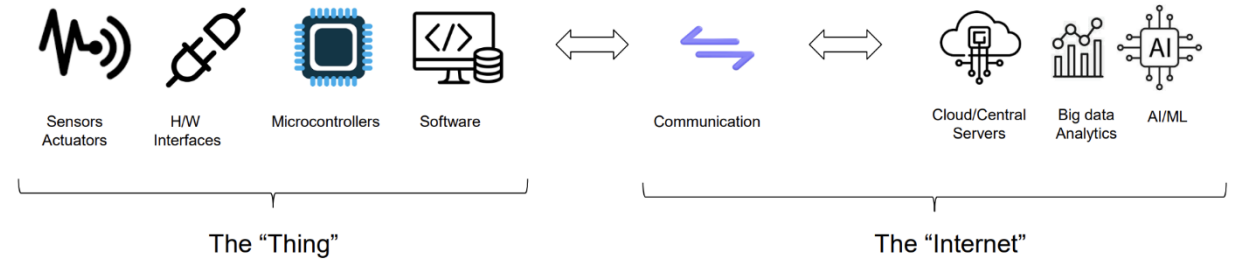
Retail & Finance
- Fuel stations
- Supermarkets
- Vending machine
- ATM/CDM/CRMs
- Self service checkouts
- Customer Relations Mgt

'Smart' devices typically have more functionality defined in software than hardware.

IoT Sub Domains



Components of an IoT System

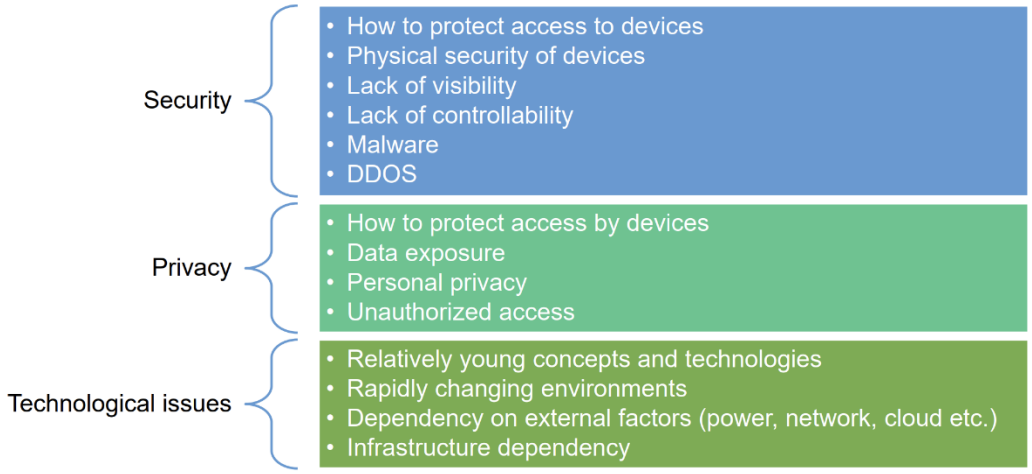


Edge Processing
Embedded Processing
The Fog Layer

- Low computing power
- Low energy
- Less reliability
- Larger volumes

- Computer vision
- Voice recognition
- Natural Language Processing
- Data Analytics
- Machine Learning

IoT Challenges

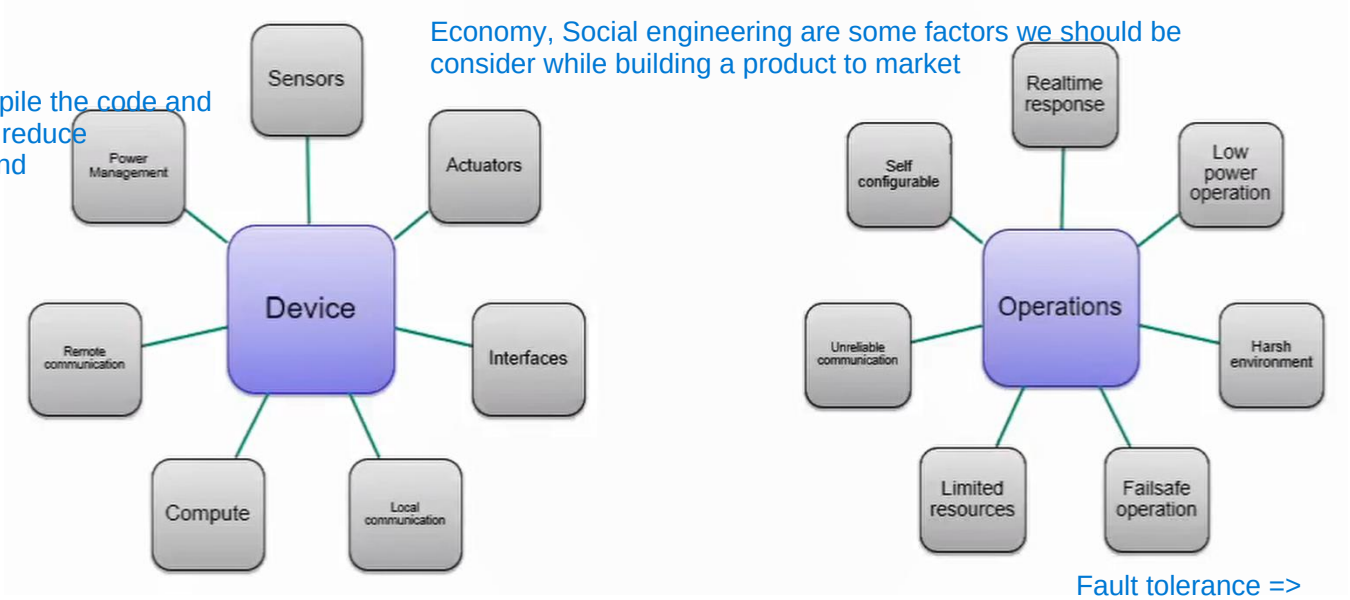


IoT => hardware and software are compact only with the required things
ex - If we don't want wifi module, we do not get a microcontroller with inbuilt wifi module

Hardware of IoT Things

We compile the code and use it to reduce power and space

Economy, Social engineering are some factors we should be consider while building a product to market



Fault tolerance =>
Do not stop even there is a failure

Fail safe =>
Fail but making sure the system remains in safe

Traditional Computers vs IoT Things

Traditional computers

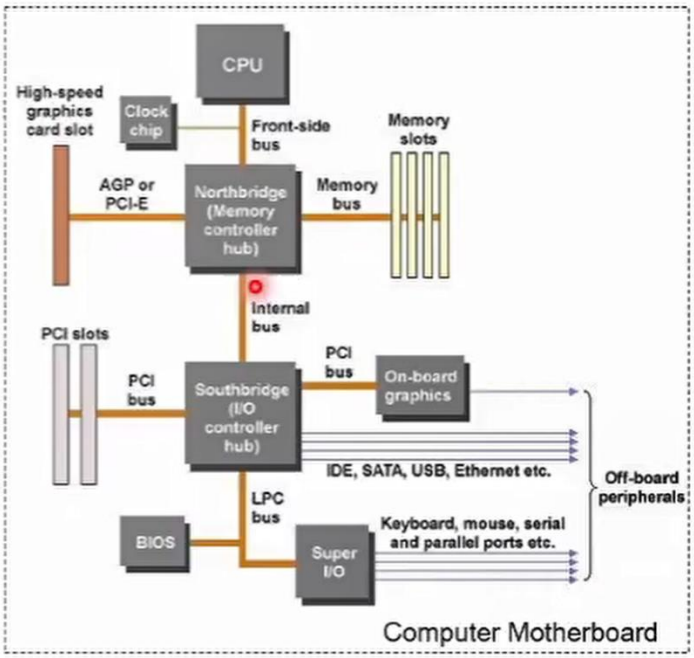
- *Designed for data-intensive computing*
  - *Larger memory & storage*
  - *Powerful, general-purpose processors*
  - *Fast network connections*
  - *Larger screens & keyboards*
- *Limited IO capabilities*
  - *Few devices like USB, Serial, Display & audio*
- *Electrical power is not an issue*

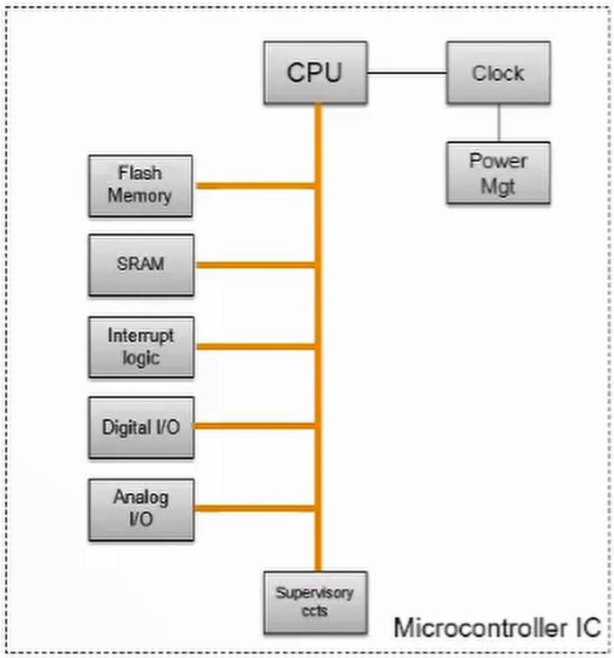
IoT Things

- *Designed for interaction with the environment*
  - *Limited memory and storage*
  - *Specialized processors* (CISC)
  - *Less reliable, slow network connections*
  - *Simple User interfaces (or headless)* → No interface at all
- *Diverse IO capabilities*
  - *Analog devices, multiple interfaces, character mode and block mode devices, analogue outputs, high-energy devices, noisy connections etc.*
- *Often has to operate on battery power with limited energy capacity*

- *The general computer architecture:*
  - *Uses multiple busses operating at different speeds to interconnect different types of devices*
  - *Use large, hierarchical memory and many IO devices*
  - *Often use buffered-IO operations*
  - *Focus on higher capacity and performance rather than energy consumption or the component count*
  - *Designed to be reliable but not necessarily fail-safe*
  - *Has many expansion features for the users*
  - *Designed to facilitate multiple purposes and software*
  - *Optimized for CISC architecture processors*

- *An IoT Thing:*
  - *Requires limited processing power to handle inputs and outputs*
  - *Requires only a small amount of faster memory*
  - *Handles only a few IO devices, but often requires a real-time response to IO events*
  - *Focus on energy saving and reliable operations*
  - *Use as few components as possible*
  - *Built with fail-safe features*
  - *Designed only to support one or a few functions*
  - *Often use RISC processors*





Microprocessors vs Microcontrollers

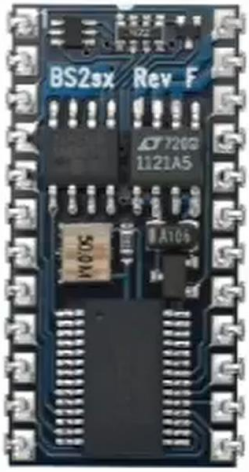
- *95W+ power dissipation*
- *5.0 GHz CPU clock*
- *64-bit memory address space*
- *32-bit IO address space*
- *Large CISC instruction set (1000+ instructions)*
- *Only the processor – need several other components to operate*
- *Optimized for data processing*

- *0.25W max power dissipation*
- *4Mhz CPU clock*
- *8K ROM & 384 byte RAM*
- *Built-in IO interfaces*
- *RISC design with only 35 instructions*
- *Built as a System-On-Chip (SoC) and no need for external components*
- *Optimized for control applications*

Popular Microcontrollers

PIC	Intel 8051	AVR	ARM
• Up to 20Mhz clock speed • Up to 16KB ROM, 512 kb RAM • No internal clock • 33 IO pins • Multiple IO peripherals	• 8 bit Intel Instruction set • 4KB ROM and 128 byte RAM • 32 IO pins • No Internal clock • Limited IO peripherals	• 8Mhz internal clock • 32 KB ROM and 32 KB RAM • 24 IO pins • Analog inputs / outputs • Many peripheral devices	• 32-bit RISC processor • External memory up to 8 GB • External clock • Limited internal IO capabilities • More like a general CPU

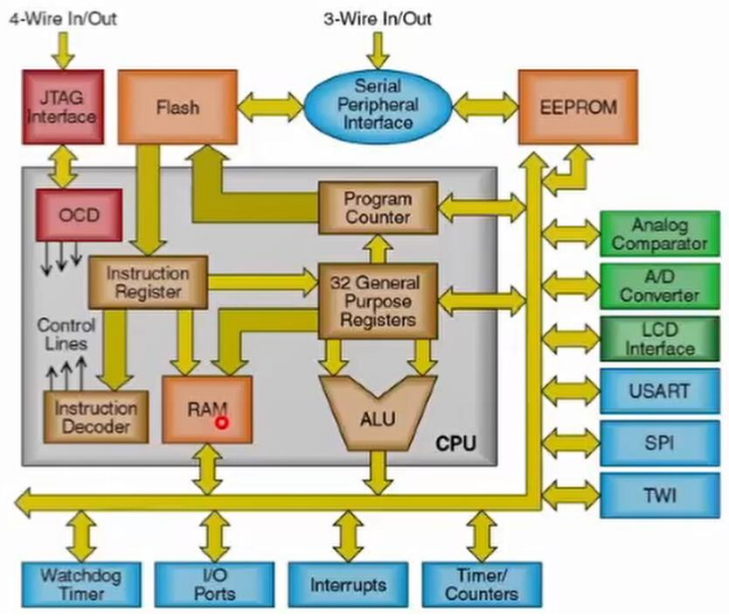
Parallax Basic Stamp



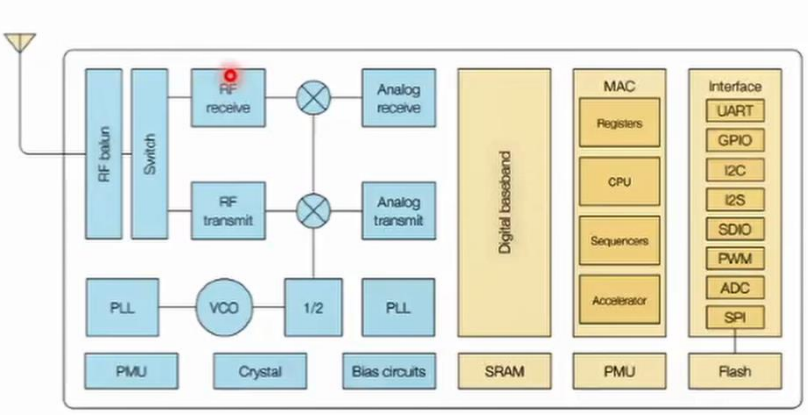
- Processor Speed: 50 MHz; ~10,000 PBASIC instructions/sec
- PBASIC Commands: 45
- Package: 24-pin DIP
- I/O pins: 16 + 2 dedicated serial
- RAM Size: 32 Bytes (6 I/O, 26 Variable)
- Scratch Pad RAM: 64 bytes
- EEPROM (Program) Size: 8 x 2 KBytes; ~4,000 PBASIC instructions
- Voltage requirements: 5.5 to 12 VDC (Vin), or 5 VDC (Vdd)
- Communication: Serial (9600 baud for programming)

Microcontroller as System-On-Chip

AVR



ESP8266



- 32-bit RISC processor
- Designed for low power consumption
- Integrates a WiFi module
- About 50kB RAM and 32kB flash memory
- Many IO options

Software of IoT Things

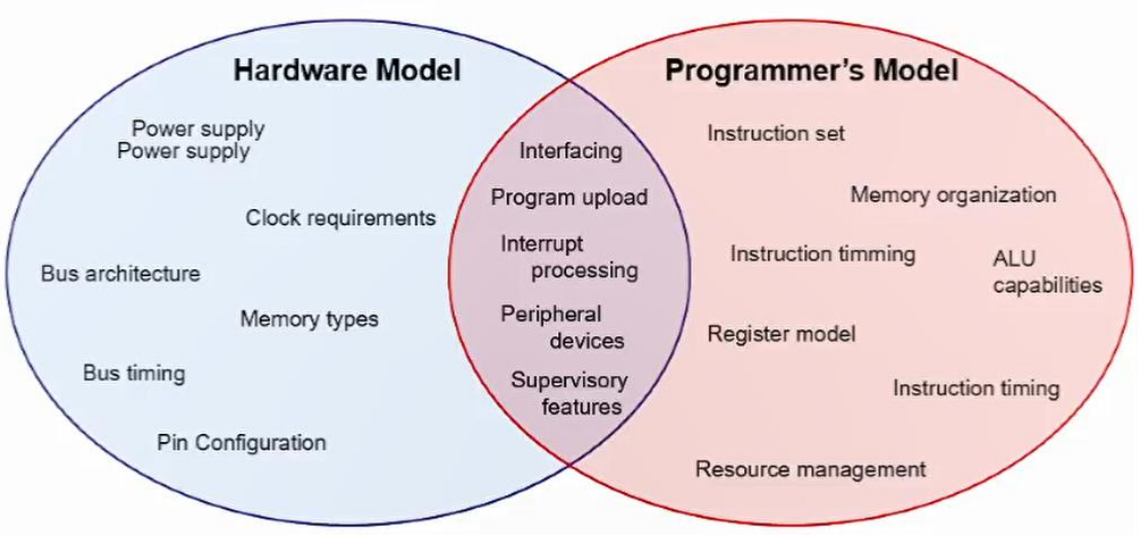
DATA DRIVEN

- *Lots of data-in-memory processing. Memory loaded with data in advance*
- *Large, fast accessible memory space*
- *Complex instruction set supporting different types of number crunching procedures*
- *Synchronous data use: Data is available in memory whenever instructions need them*
- *Internal events: Responses are mostly to internal changes*
- *Ideally suitable for stored-program serial processing*

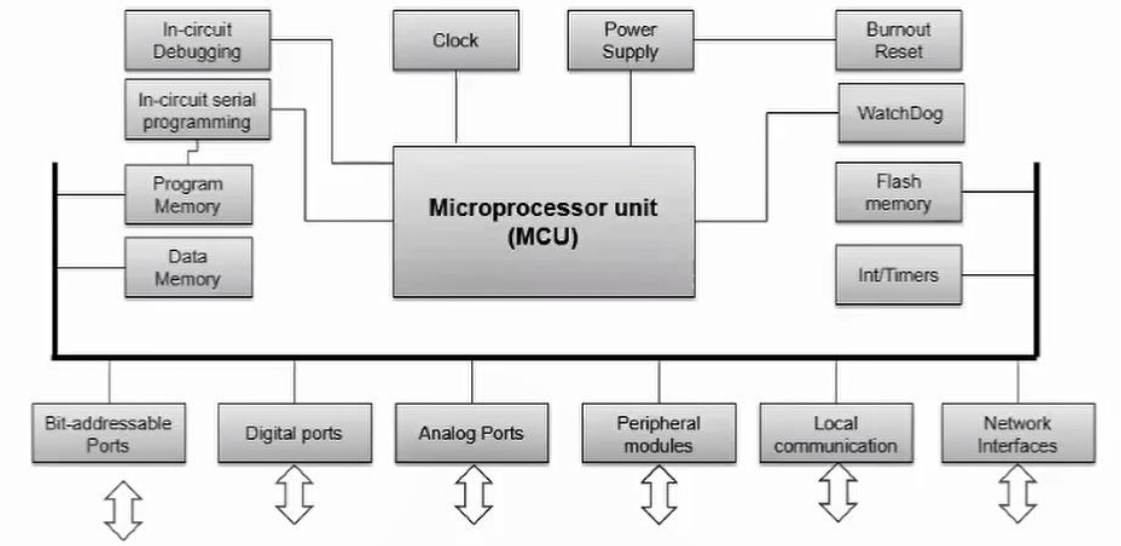
IO DRIVEN

- *Most data is in the environment (i.e. sensors) and available only when a change has occurred*
- *Limited data in memory. Speed and size not critical*
- *Asynchronous data use: Data arrived when the environment reacts, not when the algorithm needs them*
- *Simple, but fast instruction set*
- *Need to provide real-time response to external event.*
- *Need to handle multiple external events at the same time. Most requiring some immediate processing*

Software and Hardware Views



Components of a Microcontroller



MCU in Microcontroller

- *Could be a specialized design or a scaled-down version of a regular microprocessor*
  - *E.g. Intel 8048, 8051, Motorolla 68HC11*
- *Follow Harvard Architecture with the separate program and data memory*
  - *Program memory may have a larger word size – allowing long word sizes*
  - *Unified and simple timing – single clock cycle instructions*
  - *A Few RISC instructions*
- *Data memory of often viewed as a "register file" having much faster access compared to bus-based architecture*
- *Often use memory-mapped IO with built-in peripheral registers appearing as memory locations*
  - *Allow IO access like memory locations*
  - *Memory may not be continuous in some designs*
- *Program memory is usually read-only during execution*
  - *Can also be read-protected to safeguard intellectual property*
- *Provided with a separate hardware stack for subroutine calls*

Supervisory Modules

- *Monitor and ensure that the MPU functions correctly*
  - *Watch Dog Timer*
    - *Ensure that software has not hung-up in a endless loop. Causes a master reset after a timeout unless reset by software*
  - *Burnout detect*
    - *Monitor power supply and causes a master reset when a glitch / burnout is detected*
  - *Start-up delay*
    - *Delay processor execution by a pre-determined time at power-on allowing high energy devices to settle down on power requirements*
  - *Oscillator delay*
    - *Delay processor execution by a pre-defined period until the main clock oscillator becomes stable*

In Circuit Serial Programming (ICSP)

Writes the object code directly to program memory.

In Circuit Debugging (ICD)

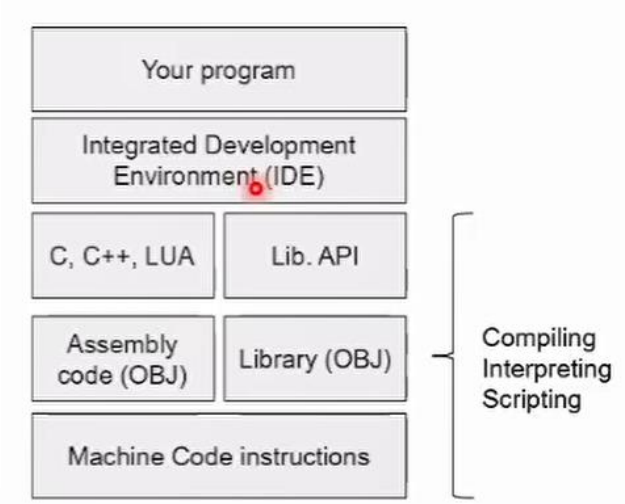
Works with the MPU providing insight into program execution.

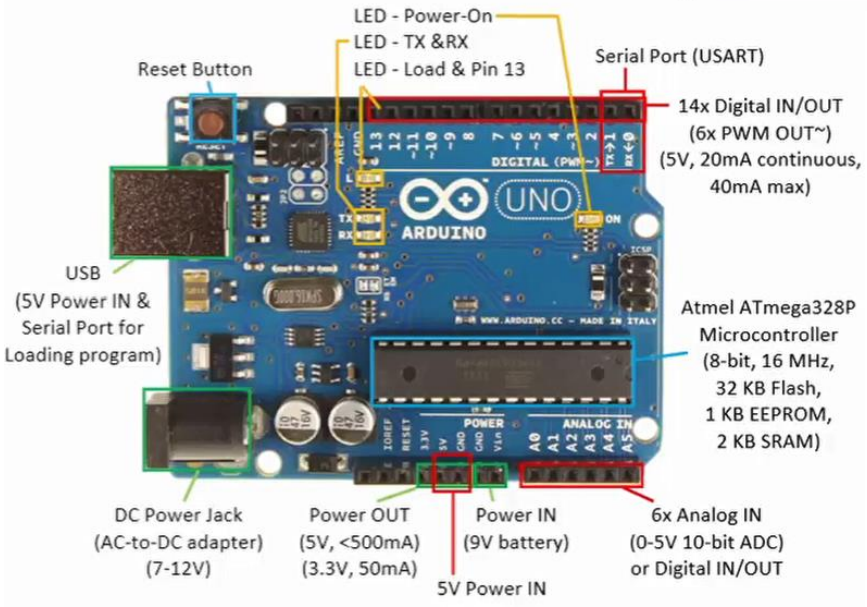
Over the Air Update (OTA)

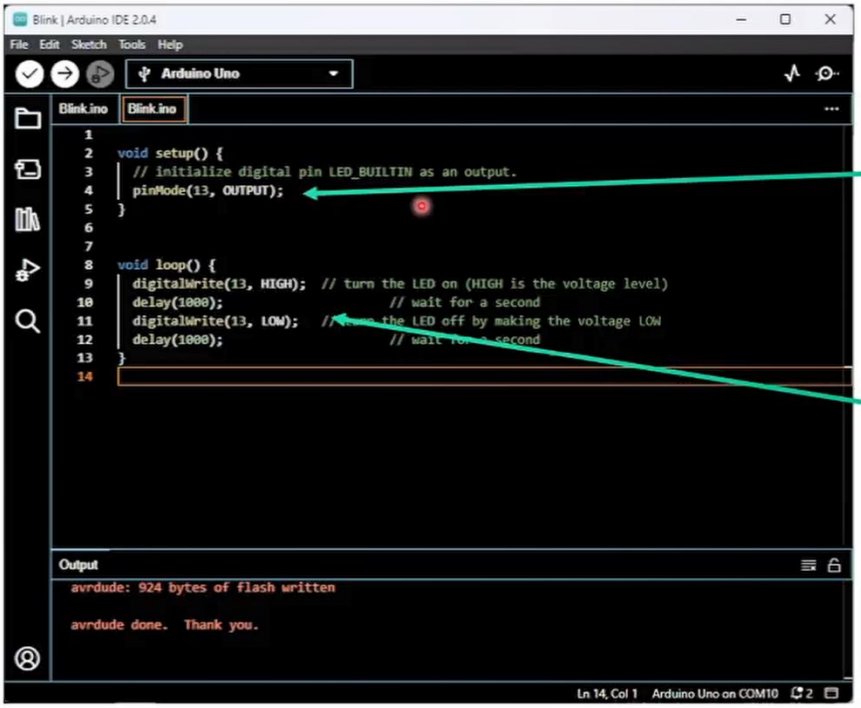
Allows firmware to be updated over a WiFi connection using bootstrap software module.

Microcontroller Programming

Eg: Blinking an LED with a set delay using Arduino







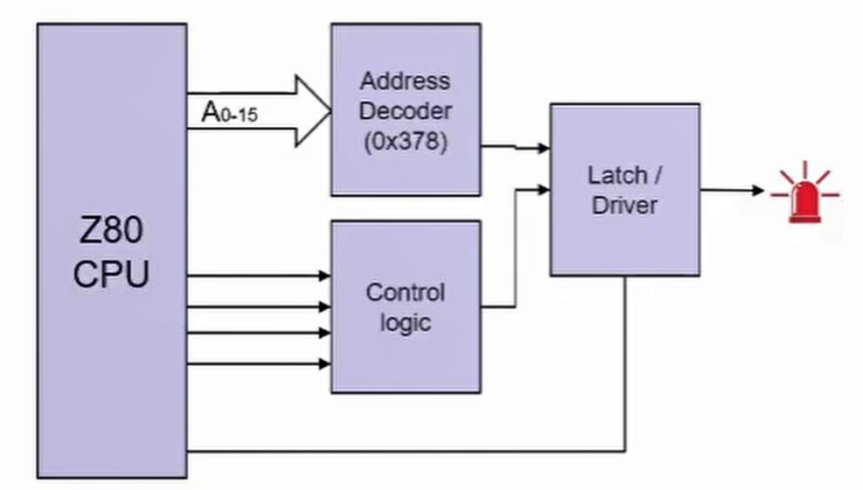
By default, all pins are set to input mode. So set pin# D13 to output mode at startup

Loop that executes continuously setting pin# D13 to high/low and waiting for 1000ms (1s) in between

It is much more complex to do the same task with a microprocessor.

```
LD A, 0x1
OUT 0x378, A
      ...

LD A, 0x0
OUT 0x378, A
```



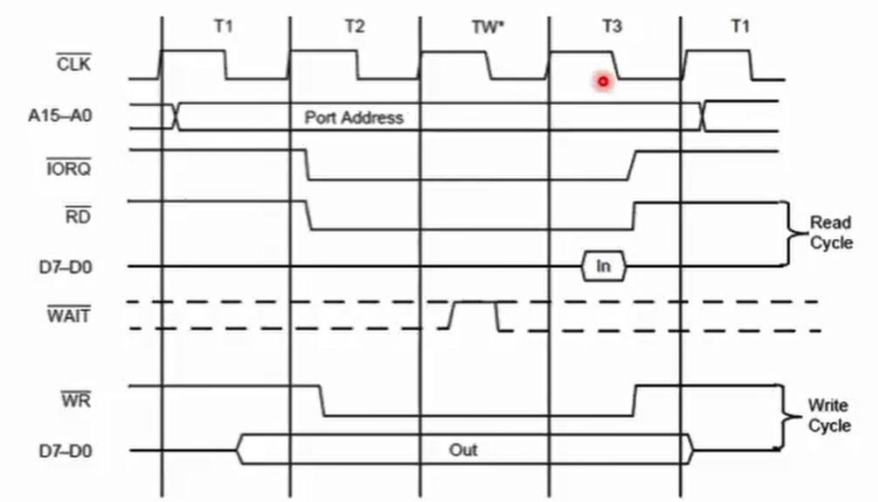
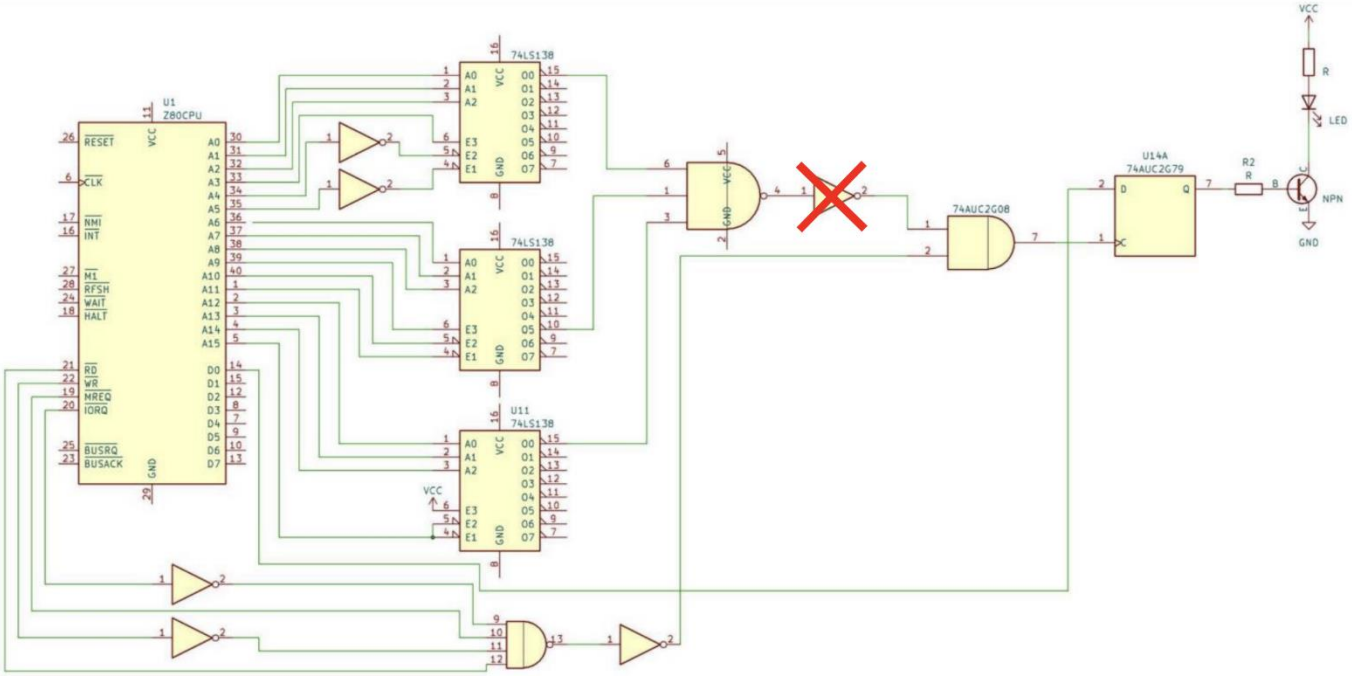
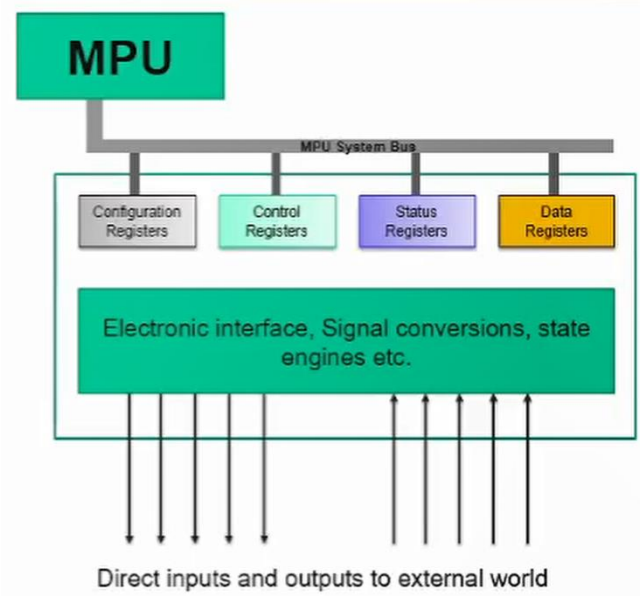


Figure 7. Input or Output Cycles



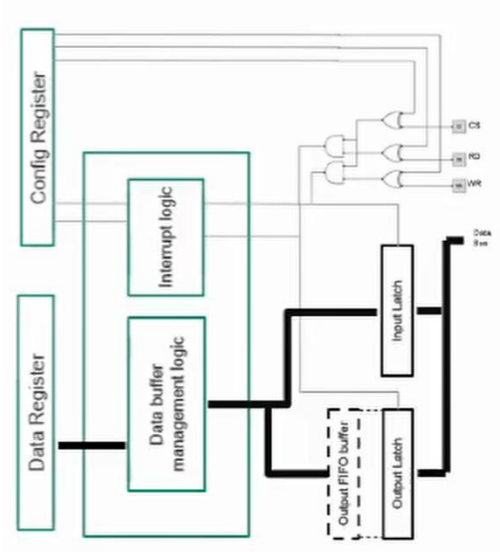
Peripheral Interfaces

- *Peripheral devices extends the hardware and functional capabilities of the MPU in a microcontroller by providing interfaces that can talk directly with the external world*
- *They can provide different functions*
  - *Digital, Analogy Input / output modules*
  - *Communication modules*
  - *Processor extensions*
  - *Supervisory / Management interfaces*
- *They provide the interface between the bus architecture of the MPU and the external world*



- *A peripheral module in general communicate with the MPU via a set of registers*
  - *These registers are mapped onto specific IO port addresses of the MPU bus*
- *Four main types of registers*
  - *Configuration registers*
    - *Typically used to provide initial configuration of the peripheral interface*
  - *Control (Command) registers*
    - *Used to send instruction on specific tasks and initiate functional operations*
  - *Status registers*
    - *Read operational status of the peripheral device including any error conditions*
  - *Data registers*
    - *Used to transfer data between the MPU and peripheral device*

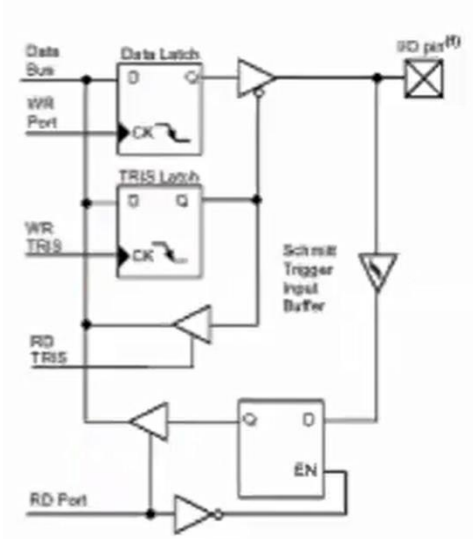
Parallel Slave Port



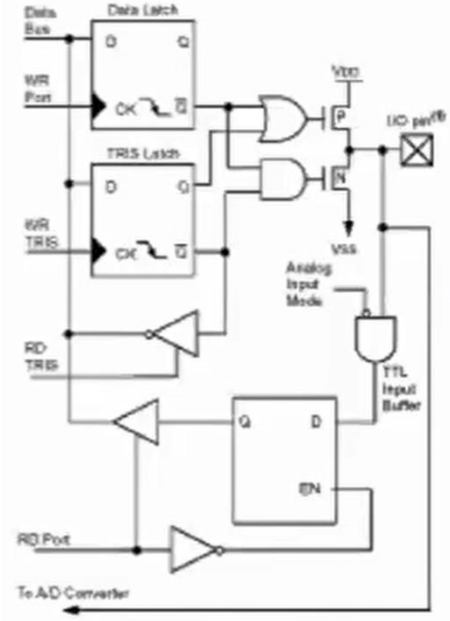
- *Used to interface with a microprocessor / system bus of an external device*
- *Compatible with control and data protocols of the external bus*
- *Configuration register provides initial setup such as enabling interrupts, polarity selection of the data latch, etc.*
- *Data output is through the write buffer register (which is usually a FIFO buffer), while read register provides the data input path*
- *Status register shows the condition of the read and write buffers*

Bit Addressable Digital IO Ports

- *Bit addressable IO ports provide direct access to individual pins of the microcontroller via their respective control / data registers*
- *At hardware level these pins can provide different functions and capabilities, often controlled by the configuration register*
  - *Simple digital input / output with data latching*
  - *Combined analog / digital IO*
  - *Additional physical layer properties such as*

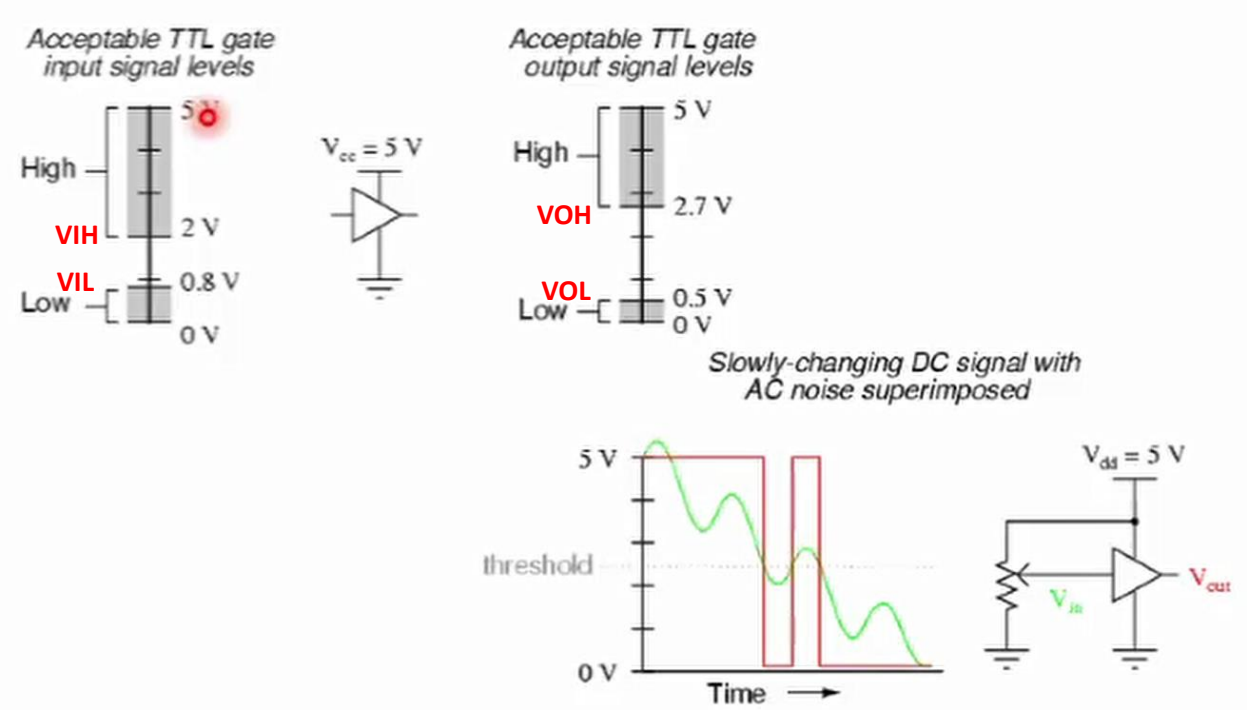


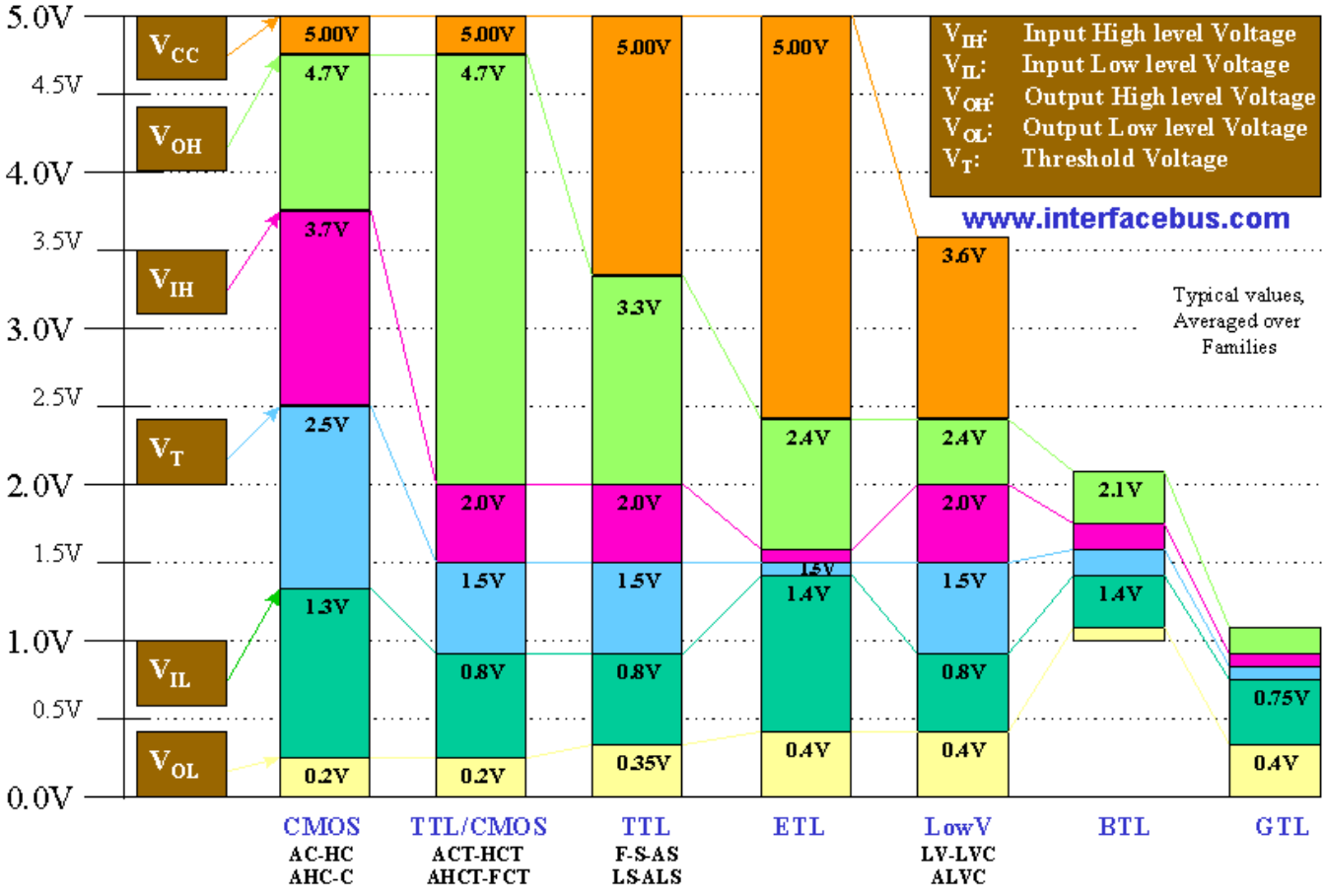
- *Simple Bit-addressable digital IO port in PIC16XXX series*
- *Configured via TRIS register that determine whether the individual bit act as an Input pin or as a latched output pin*
  - *Input pin when TRIS is true*
- *Has Schmitt trigger buffers that implement a hysteresis band to support noisy inputs*
- *Input can be latched or in transparent mode via an external **RDPort** signal*
- *The same configuration is repeated for all 8 bits in the IO port (PORTD)*



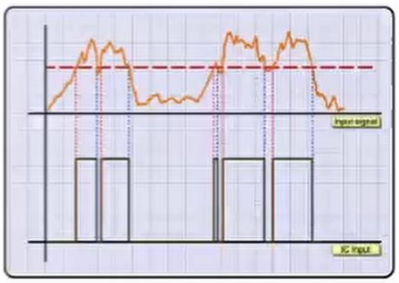
- *Digital bit addressable port with high-current capability and analog input support (PORTA in PIC16XXX)*
- *Configured via two registers*
  - *TRIS for direction control*
  - *Analog Input Mode – enable analog input*
- *Two MOSFET at the output stage allow high current driving and sinking*
- *Selecting Analog input mode disable digital reading*
  - *Pin must be in input mode to support analog input mode*
  - *Separate AtoD converter (not shown in the schematic) is required*

Digital IO – Voltage Levels

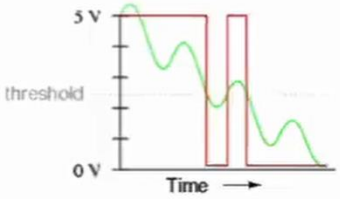




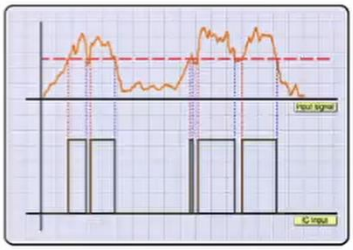
Digital IO – Schmitt Trigger Inputs



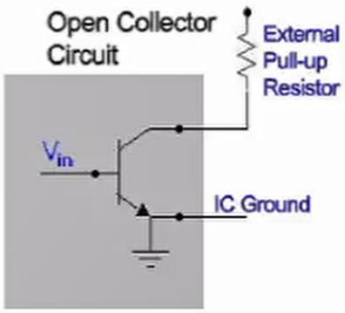
- *Use of single threshold is problematic*
  - *Threshold may not be consistent*
  - *Signal may cross threshold due to noise*
- ***Solution**: Use different thresholds for Low -> High and High - > Low transitions*

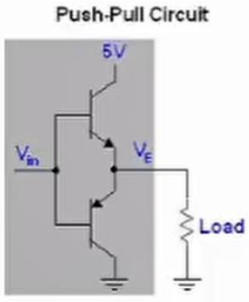


- *Use of different thresholds avoid the problem of prohibited range values*

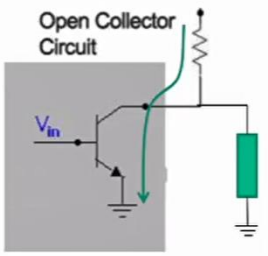


Digital IO – Push-Pull vs Open Outputs

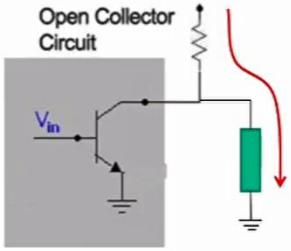




- *An open (collector / drain) configuration provides active driving only on one logic state*
- *External pull-up resisters are needed to drive the load when the device output is on "Open" state*
  - *Loading on the resister must be carefully considered to ensure correct logic state at the output*

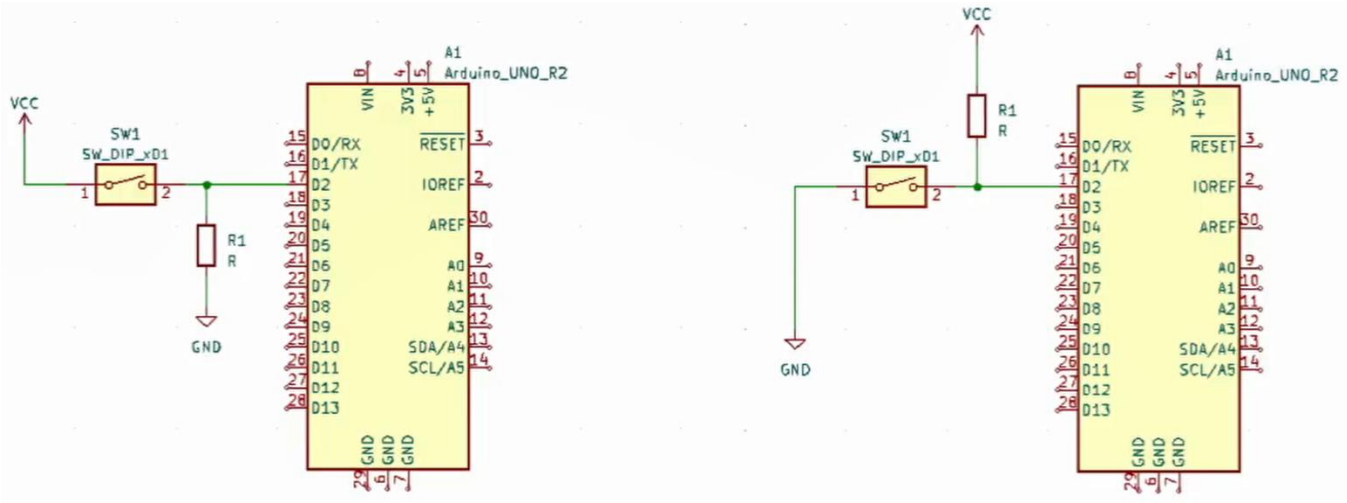


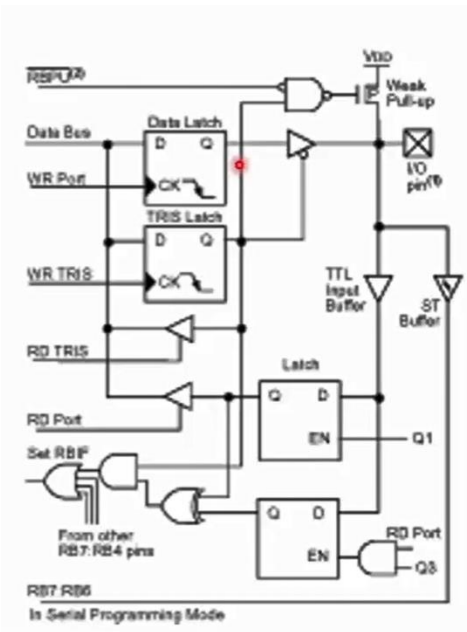
When the output is at Logic –Low state the load voltage is clamped at Vce of the output transistor. The external pull-up resister act as a current limiter



When the output is at Logic–High state current flows through the load. Output voltage is determined by potential division between the load and the external pull-up resister. Care must be taken to maintain output voltage at logic high level.

Digital IO – Pull-Up vs Pull-Down





- *Digital IO port with open collector (drain) outputs, programmable pull up resisters and interrupt capability (PORTB in PIC16XXX)*
- *Configured via*
  - *TRIS register*
    - *Set pin direction (input , output)*
  - *PULLUP*
    - *Enable / Disable internal weak pull-up resisters*
  - *INTCONF*
    - *Enable interrupt on pin change*
- *Open collector (drain outputs) with low current sink capabilities*
- *Also support schimmitt trigger inputs to other peripheral devices when not used for digital IO*

Toggle Button

LED_Toggle_Cont | Arduino IDE 2.1.0

File Edit Sketch Tools Help

Arduino Uno

LED_Toggle_Cont.ino

```
1  void setup() {
2    pinMode(2,INPUT);
3    pinMode(13, OUTPUT);  // Internal LED pin
4  }
5
6  bool LED_On = false;
7
8  void loop() {
9    int pin = digitalRead(2);
10   if (pin==1)
11     LED_On = !LED_On;
12
13   if (LED_On)
14       digitalWrite(13,HIGH);
15     else
16       digitalWrite(13,LOW);
17 }
18
```

What's wrong here…?

LED_Toggle_State | Arduino IDE 2.1.0

File Edit Sketch Tools Help

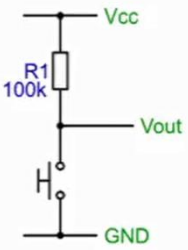
Arduino Uno

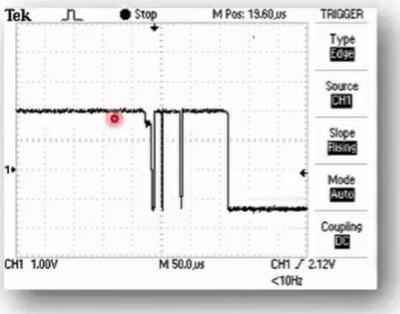
LED_Toggle_State.ino

```
1  void setup() {
2    pinMode(2,INPUT);
3    pinMode(13, OUTPUT);  // Internal LED pin
4  }
5
6  bool LED_On = false;
7  bool Is_Btn_Press = false;
8
9  void loop() {
10   int pin = digitalRead(2);
11   if (pin==1) {
12     if (!Is_Btn_Press){   // rising edge
13       LED_On = !LED_On;
14       Is_Btn_Press = true;
15     }
16   } else {
17     Is_Btn_Press=false;  //Button released
18   }
19
20   if (LED_On)
21       digitalWrite(13,HIGH);
22     else
23       digitalWrite(13,LOW);
24 }
25
```

Switch Debouncing

- *Pressing or releasing a mechanical switch in general does not produce a clean edge in the output waveform*
  - *Digital input may interpret this as multiple edges*
  - *Referred to a "Bouncing" on the input*

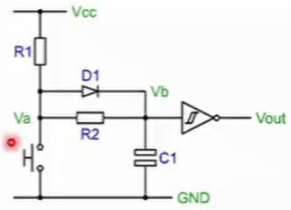


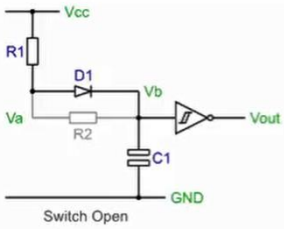


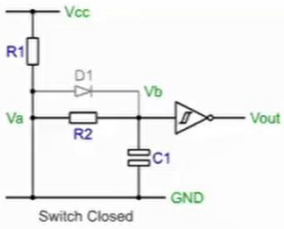
Debouncing can be done in hardware or using software.

LED_Toggle_State.ino

```
1  void setup() {
2    pinMode(9,INPUT);
3    pinMode(13, OUTPUT);  // Internal LED pin
4  }
5
6  bool LED_On = false;
7  bool Is_Btn_Press = false;
8
9  void loop() {
10   int pin = digitalRead(9);
11   if (pin==1) {
12     if (!Is_Btn_Press){   // rising edge
13       LED_On = !LED_On;
14       Is_Btn_Press = true;
15     }
16   } else {
17     Is_Btn_Press=false;  //Button released
18   }
19
20   if (LED_On)
21       digitalWrite(13,HIGH);
22     else
23       digitalWrite(13,LOW);
24   delay(1);
25 }
26
```







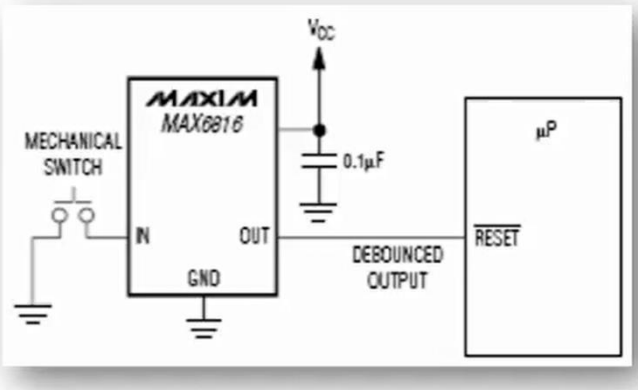
- *Switch Open*
  - *The capacitor C1 will charge via R1 and D1.*
  - *In time, C1 will charge and Vb will reach within 0.7V of Vcc.*
  - *Therefore the output of the inverting Schmitt trigger will be a logic 0.*
- *Switch Close*
  - *The capacitor will discharge via R2.*
  - *In time, C1 will discharge and Vb will reach 0V.*
  - *Therefore the output of the inverting Schmitt trigger will be a logic 1.*

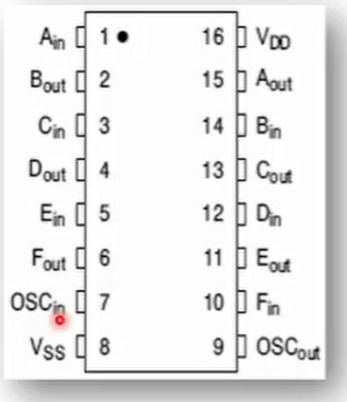
COUNTER METHOD

- *Setup a counter variable, initialize to zero.*
- *Setup a regular sampling event, perhaps using a timer. Use a period of about 1ms.*
- *On a sample event:*
- *if switch signal is high then*
- *Reset the counter variable to zero*
- *Set internal switch state to released*
- *else*
- *Increment the counter variable to a*
- *maximum of 10*
- *end if*
- *if counter=10 then*
- *Set internal switch state to pressed*
- *end if*

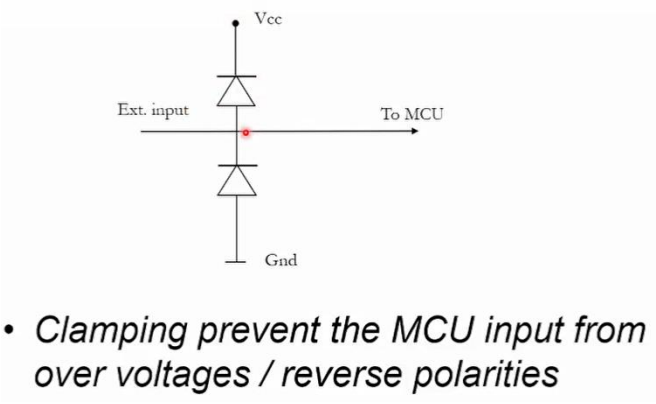
SHIFT REGISTER METHOD

- *Setup a variable to act as a shift register, initialise it to xFF.*
- *Setup a regular sampling event, perhaps using a timer. Use a period of about 1ms.*
- *On a sample event:*
- *Shift the variable towards the most*
- *significant bit*
- *Set the least significant bit to the*
- *current switch value*
- *if shift register val=0 then*
- *Set internal switch state to pressed*
- *else*
- *Set internal switch state to released*
- *end if*





Protecting Inputs



- *Clamping prevent the MCU input from over voltages / reverse polarities*