University of Nottingham
UK | CHINA | MALAYSIA

Lecture 4

State Tables, Finite State Machines and Interrupt

Mechatronics

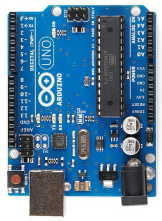
MMME3085

Module Convenor – Abdelkhalick Mohammad

Lecture Objectives

- To introduce two simple approaches to the programming of **sequences of events**
- To introduce the concepts of:
  - **State Tables**
  - **Finite State Machines**
- To identify some **applications** of the finite state machine architecture
- To understand the concept on **Interrupt** in microprocess and how they can be implemented on Arduino
- **Laboratory 1**: Signals and Sensors – Link to Lectures

A typical Mechatronics System



Computer or micro-processor

2,3,10

1,4,10

Program e.g., in C

Buses in computer

Digital output inter-face; timer

Digital input inter-face; counters

Digital signal

DAC

Analogue signal

Digital signal

5,6

Digital signal

ADC

Analogue signal

Digital signal

7,8,9

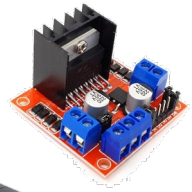
Electronic hardware

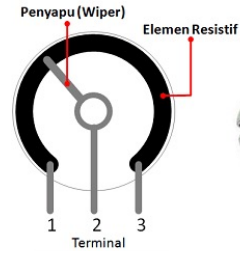
Actuators

Mechanical system

Sensors

Electronic hardware






University of
Nottingham
UK | CHINA | MALAYSIA
Recap

Dealing with "Train of pulses" signal … Timer/Counters

Last time we looked at

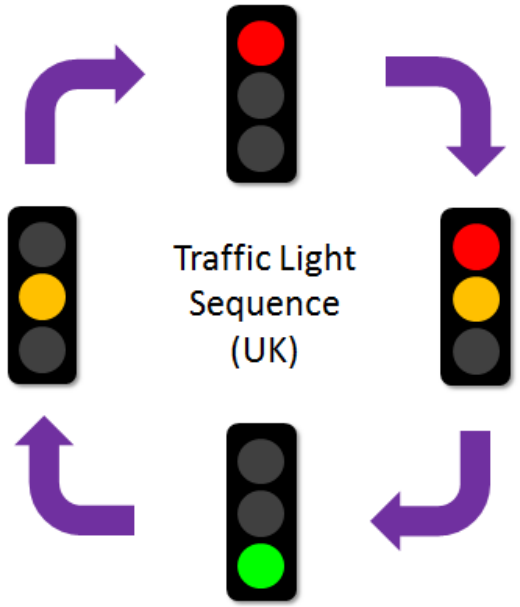
- The challenges in using **simple interface** (e.g., digital read/write) in counting events or measuring/generating frequencies
- Why **hardware Timer/Counters** (T/Cs) can be a solution for such challenge.
- How to **configure** the function of T/Cs using registers
- **Applications** of T/Cs such as frequency generation, counting high frequency pulses and generating PWM.
- Using **datasheet** of the Arduino to search configure T/Cs.

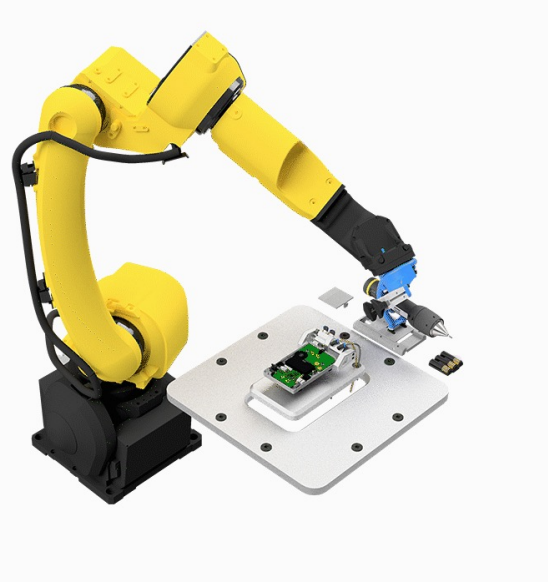
Sequences and States

Why are we interested in sequence?!

Mechatronic systems are often required to perform tasks in a *sequence*, either *one after the other* or based on *external inputs or events*.

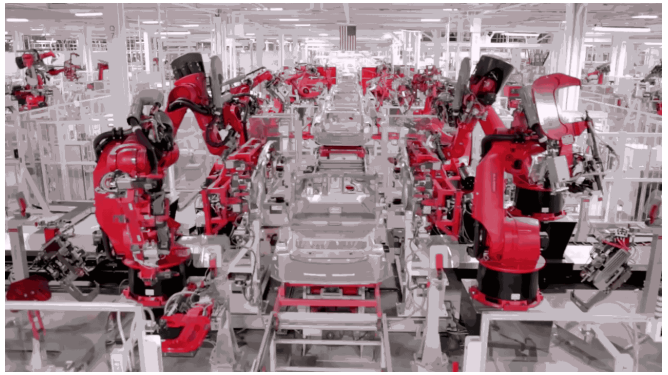
- *Traffic lights*: The controller needs to turn the traffic lights in *sequence* waiting between the switching for a specific time to elapse.
- Assembly robot: The robot needs to step through a sequence of tasks one after the other, waiting between tasks for the previous task to complete. The robotic arm would also need to be able to *respond to external inputs*, such as sensors that detect the presence of parts or tools.





Why are we interested in sequence?!

- We can easily end up with *messy code* when handling sequences and decisions of *complex systems*.
- In the "bad old days" of early programming, "*spaghetti code*" was a major issue. Spaghetti code is code that is difficult to read and understand because it is poorly structured with tangled logic.
- *Frameworks* can help us to avoid spaghetti code by providing an organized way to handle sequences and decisions.
- Frameworks can also help us to avoid *common mistakes*, such as forgetting to handle all possible inputs or outputting incorrect values.
- There are several different frameworks such as:
  - **State machines**
  - **Decision trees**
  - **Rule engines**



University of Nottingham
UK | CHINA | MALAYSIA

Programming Vocabulary

One-dimensional (1D) Array

- You should be familiar with the concept of an ***array*** in Matlab: a *block of variables* with each element identified by an ***index***
- In C the syntax of a **1D** array is:

```
int myArray[arraySize];
```

- Or if we are declaring and initialising at the same time:

```
int myArray[arraySize]{10,20,30,40};
```

- We access each array element as follows:

```
myArray[index] = 99;
```

- For example

```
myArray[0] equals to 10
myArray[1] equals to 20
```

Two-dimensional (2D) Array

- In practice we often need a 2D array which is effectively *a table of numbers* or *a **matrix***:

```
int my2DArray[firstSize][secSize]
                    {{10,20},
                    {30,40},
                    {50,60}};
```

- This is indexed using separate square brackets:

```
my2DArray[rowNum][colNum] equals to 51;
```

Enumerated Constants

- We may have a limited set of situations each identified by an integer e.g., 0, 1, 2, 3 etc.
- But it may be easiest to *give these numbers names,* so it makes obvious what they refer to
- Use "enumerated constants" e.g.

```
enum daysOfWeek {SUNDAY, MONDAY, TUESDAY,
        WEDNESDAY, THURSDAY, FRIDAY, SATURDAY};
```

  which defines **SUNDAY** as *0*, **MONDAY** as *1* etc.
- Trying to set a variable of type **daysOfWeek** to (for example) 10 will give error & trap a bug!

"switch" Statement

- Sometimes we have to decide between some clearly defined situations, for example associated with different numbers
  - Could use a rather complex if statement or…
  - Use a "switch" statement:

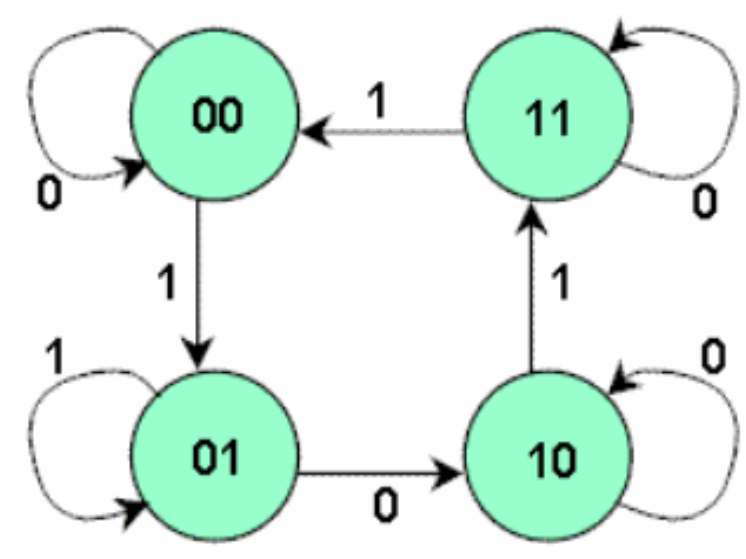
```
enum daysOfWeek {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY};
int i = 0; // Number of the week days
void setup() {
Serial.begin(9600);
}
void loop() {
for (i = 0; i < 7; i++){
int day=daysOfWeek(i); //if it is Sunday, day will be 0, if Monday it will be 1 and so on
Serial.print("Today is ");
switch (day)
{
case SUNDAY:
Serial.print("SUNDAY: Go for Swiming");
break;
case MONDAY:
Serial.print("Monday: Go to Computer Engineering Lecture");
break;
case TUESDAY:
Serial.print("Tuesday: Go to Computer Engineering Laboratory");
break;
case WEDNESDAY:
Serial.print("Wednesday: Studay Mechatronics and Computer Engineering");
break;
case THURSDAY:
Serial.print("Thursday: Go to Mechatronics Lecture");
break;
case FRIDAY:
Serial.print("Friday: Go to Mechatronics Seminar");
break;
case SATURDAY:
Serial.print("Saturday: Take a rest");
break;
}
Serial.print('\n');
delay(2000);
}}
```

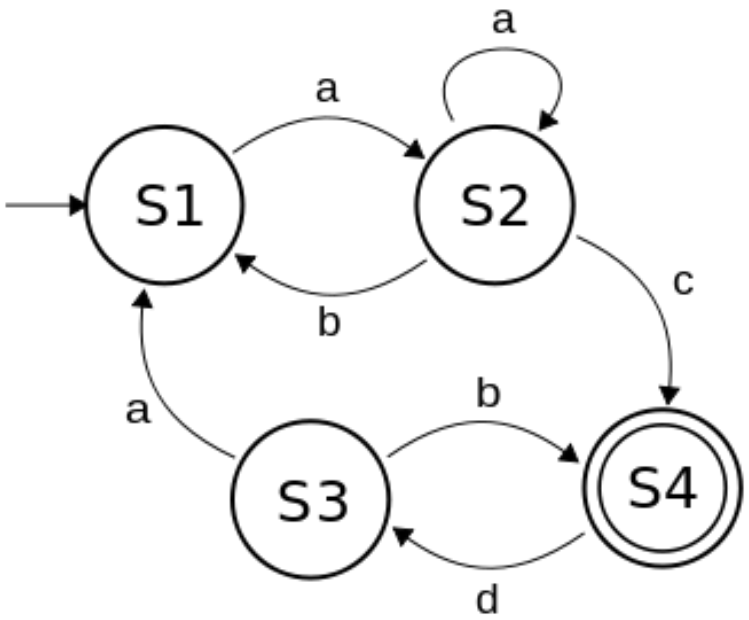
Today is SUNDAY: Go for Swiming
Today is Monday: Go to Computer Engineering Lecture
Today is Tuesday: Go to Computer Engineering Laboratory
Today is Wednesday: Studay Mechatronics and Computer Engineering
Today is Thursday: Go to Mechatronics Lecture
Today is Friday: Go to Mechatronics Seminar
Today is Saturday: Take a rest
Today is SUNDAY: Go for Swiming
Today is Monday: Go to Computer Engineering Lecture

State Tables and Finite State Machines

Let us consider two situations:

- **Straightforward sequences** where tasks are executed *one after the other*. This can be easily implemented with state tables.
- **More complex situations** where order varies *on-the-fly* depending on inputs: need more complex solution (finite state machine).





University of Nottingham
UK | CHINA | MALAYSIA

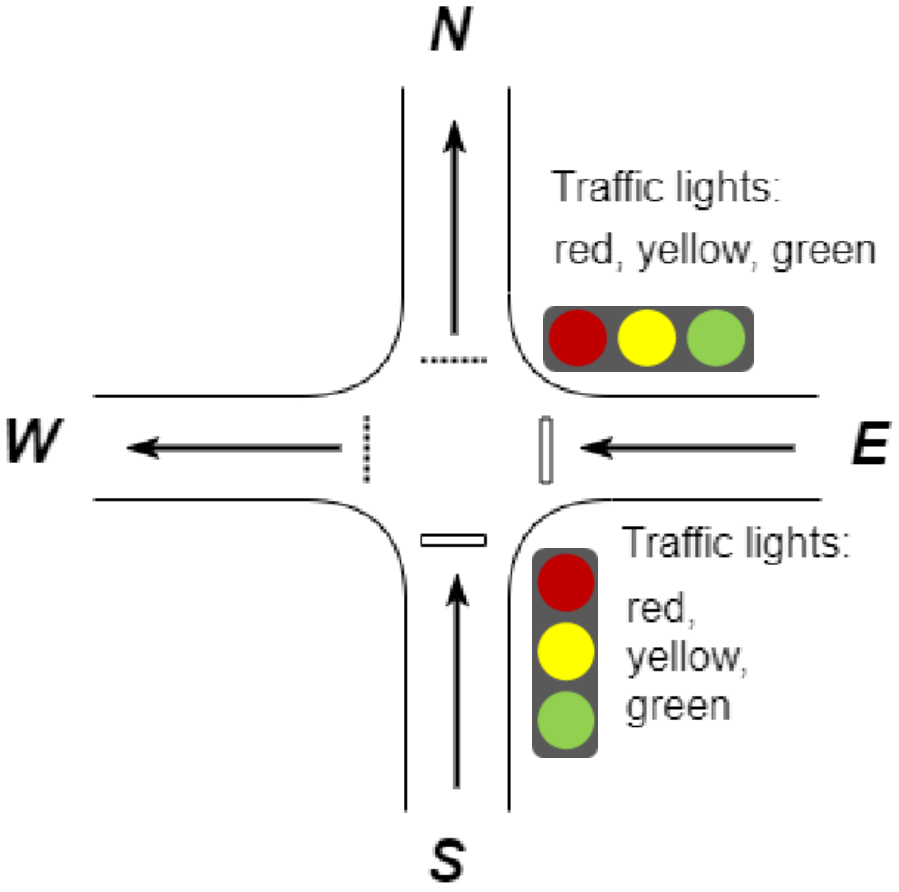
State Tables

State Table Example – Traffic Lights

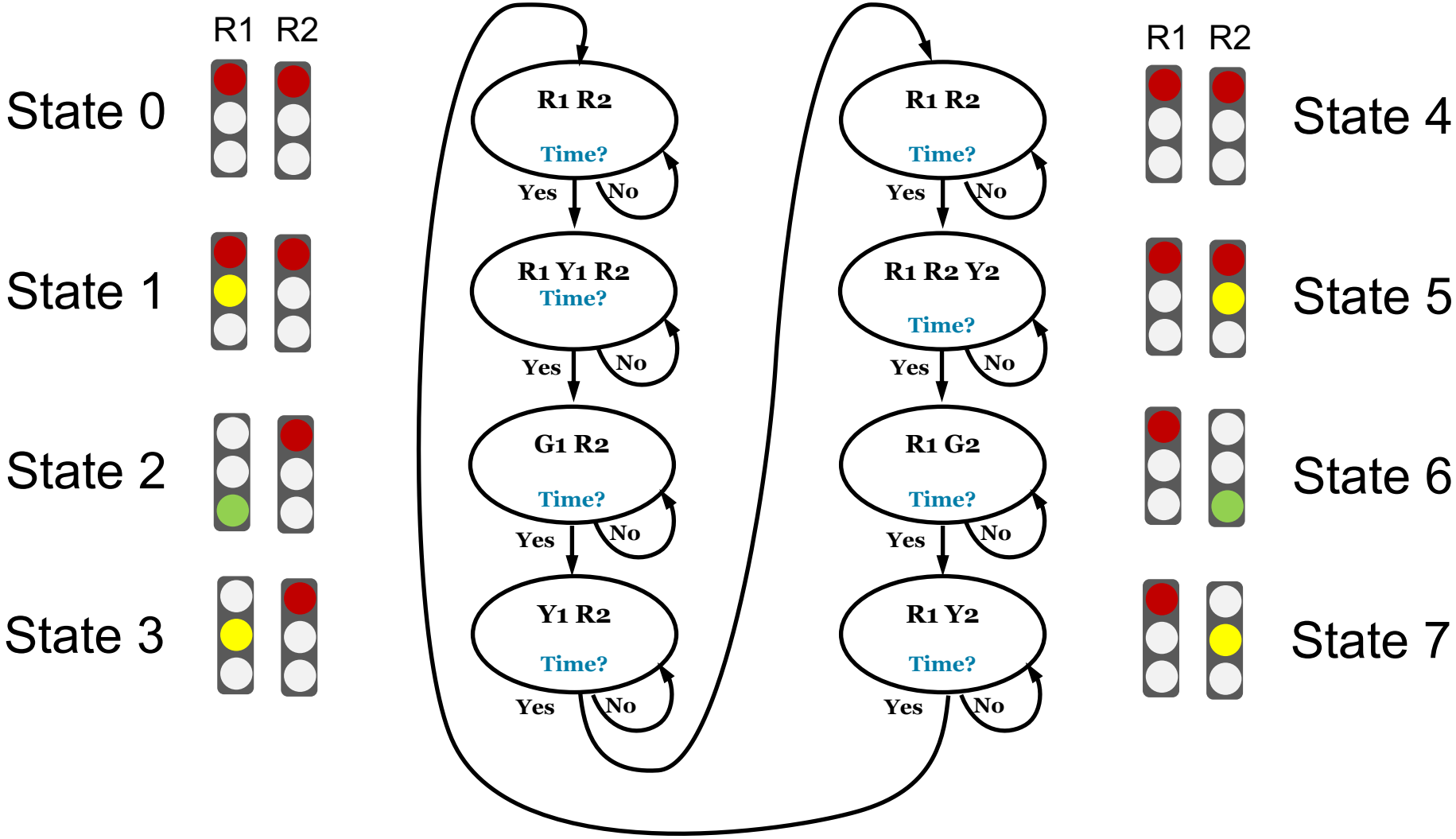
- Example: Simple traffic lights at junction

Simple traffic lights at a junction are typically controlled by a *single set of lights for each direction* of traffic. The lights can be red, green, or amber (yellow).

- Red means stop.
- Amber means stop if it is safe to do so.
- Green means you can go, but you must still give way to pedestrians and other vehicles that have the right of way.

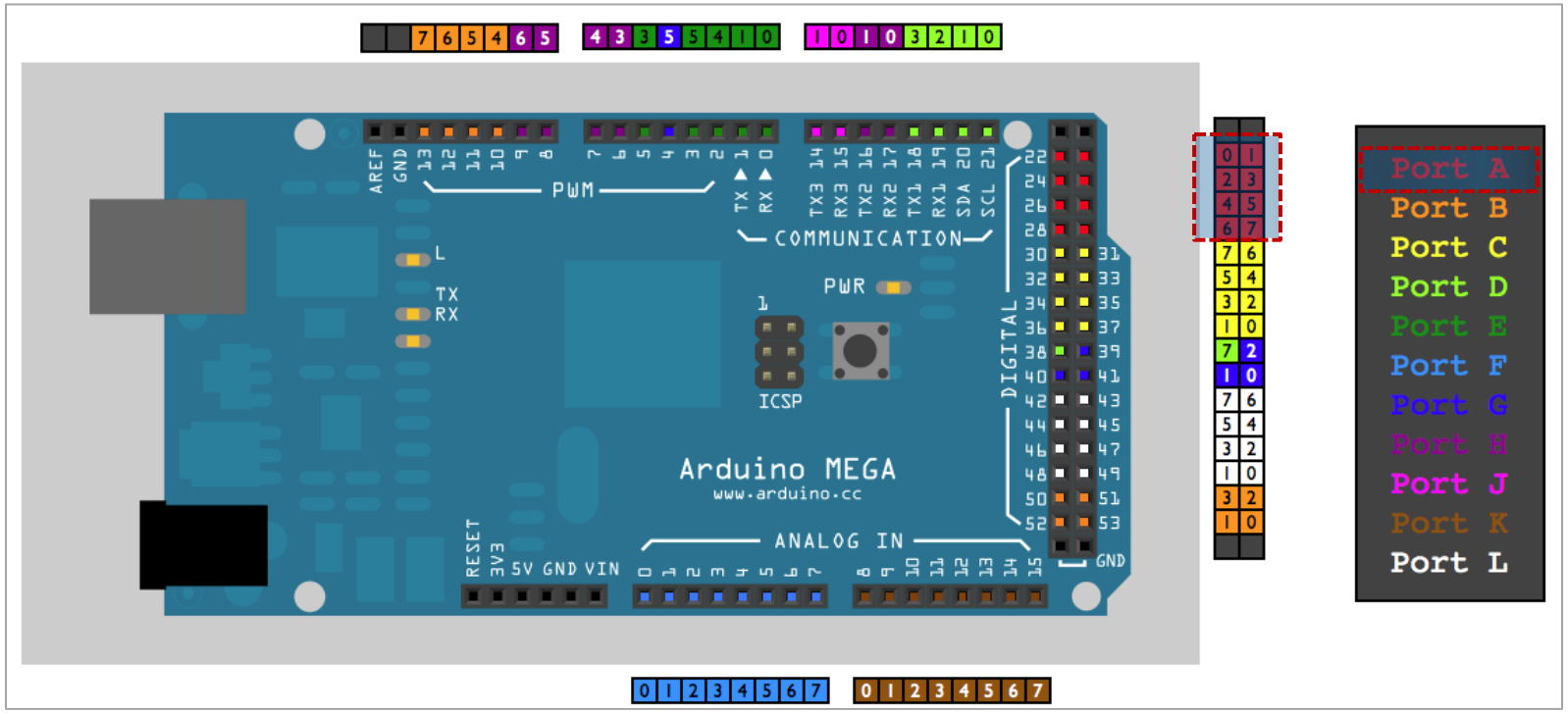


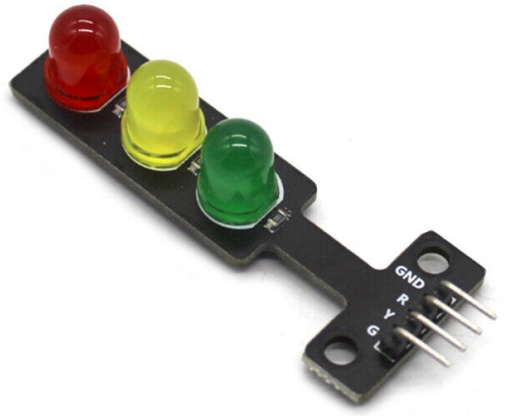
State Table Example – Traffic Lights



State Table Example – Traffic Lights

This can be implemented using the Arduino and a board with LEDs or connecting individual LEDs (and resistors) on a breadboard





Traffic Light Single Control Panel 3 Colour LED Module 5V For Arduino

PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0

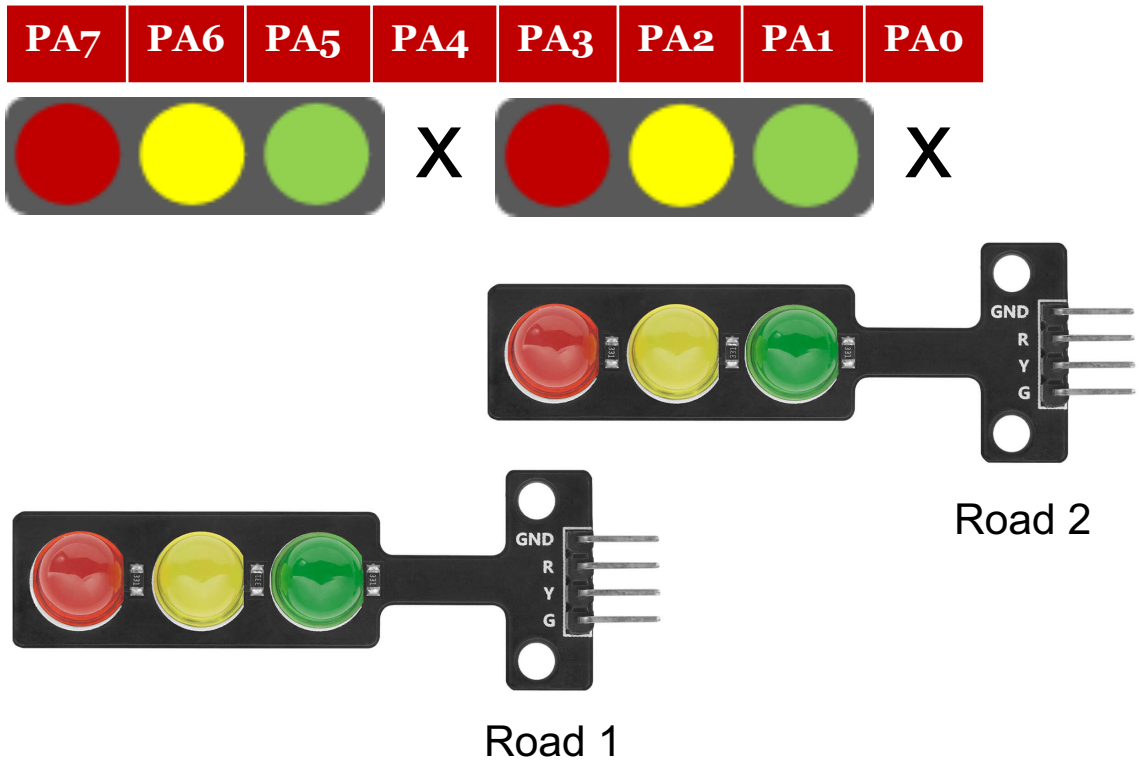
X

X

State Table Example – Traffic Lights

This can be implemented using the Arduino and a board with LEDs or connecting individual LEDs (and resistors) on a breadboard

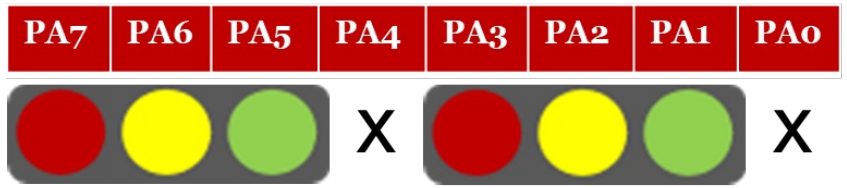
- Use bits 1-3 (Road 2) and 5-7 (Road 1) to simulate timed phasing at a crossroads
- One way to program this is to construct a very simple table of containing the states corresponding to each step in the sequence (i.e., *state table*)
- Can then convert *array of bits* into binary numbers
- These numbers then converted to *hexadecimal numbers*.
- We write a code to send these Hex codes to PORTA *one by one* in the same sequence.



State Table Example – Traffic Lights

This can be implemented using the Arduino and a board with LEDs or connecting individual LEDs (and resistors) on a breadboard

- Use bits 1-3 (Road 2) and 5-7 (Road 1) to simulate timed phasing at a crossroads
- One way to program this is to construct a very simple table of containing the states corresponding to each step in the sequence (i.e., *state table*)
- Can then convert array of bits into binary numbers
- These numbers then converted to hexadecimal numbers.
- We write a code to send these Hex codes to PORTA one by one in the same sequence.



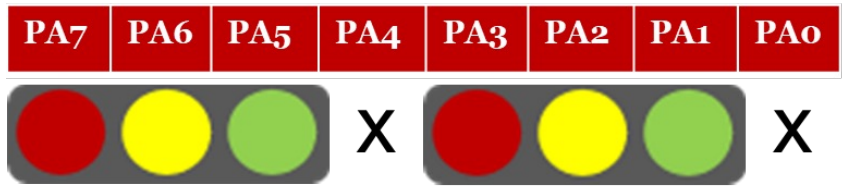
								Hex	Time
1	0	0	0	1	0	0	0	88	1
1	1	0	0	1	0	0	0	C8	2
0	0	1	0	1	0	0	0	28	5
0	1	0	0	1	0	0	0	48	2
1	0	0	0	1	0	0	0	88	1
1	0	0	0	1	1	0	0	8C	2
1	0	0	0	0	0	1	0	82	5
1	0	0	0	0	1	0	0	84	2

State Table Example – Traffic Lights

```
const int NumStates = 8;
int state = 0;
byte StateTable [NumStates][2]={
 {0x88, 1},
 {0xC8, 2},
 {0x28, 5},
 {0x48, 2},
 {0x88, 1},
 {0x8C, 2},
 {0x82, 5},
 {0x84, 2}};

void setup() {
 DDRA = 0xFF; // Set PORTA as output
Serial.begin(9600); }

void loop() {
Serial.print(state);
Serial.print(' ');
PORTA = StateTable[state][0];
Serial.print(PORTA, HEX);
Serial.print(' ');
Serial.print("Waiting Time is:");
Serial.print(StateTable[state][1]);
Serial.print('\n');
delay(StateTable[state][1]*1000);

if(state < NumStates-1)
{
state++;
}
else
{
  state = 0;
}
}
```

Code output

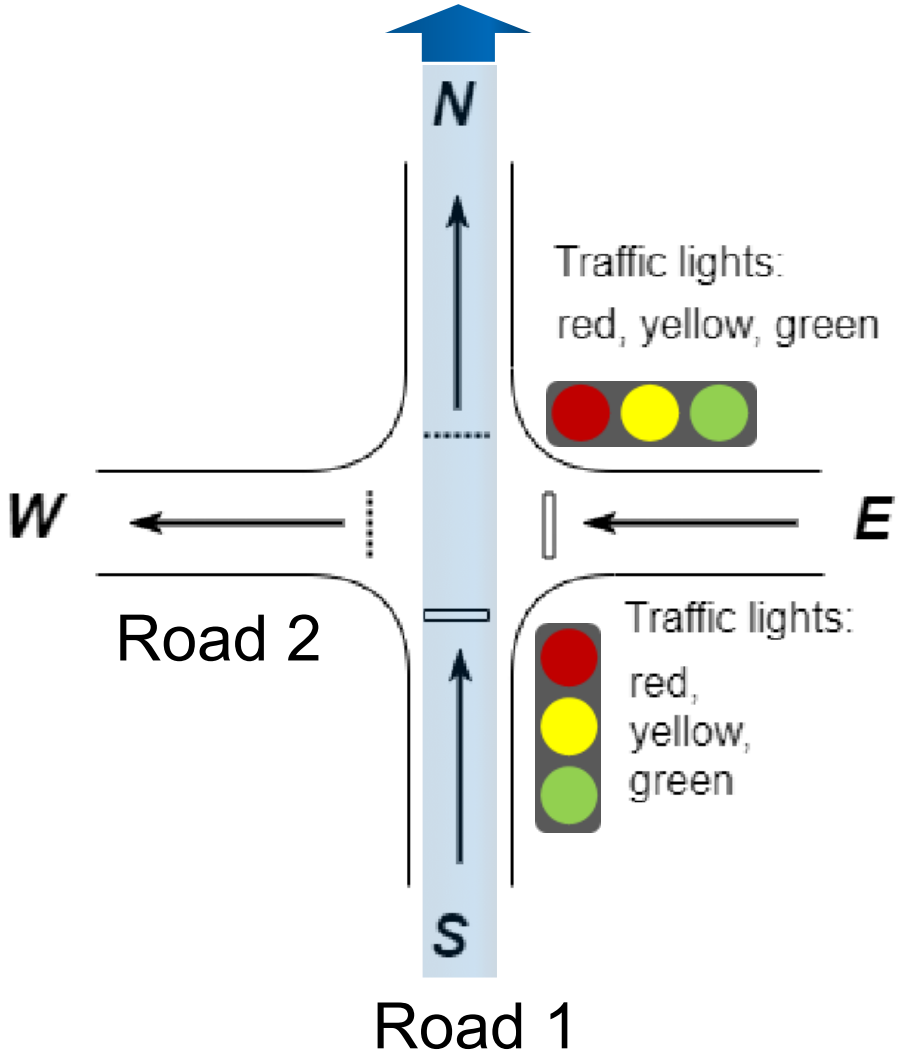
0 88 Waiting Time is:1
1 C8 Waiting Time is:2
2 28 Waiting Time is:5
3 48 Waiting Time is:2
4 88 Waiting Time is:1
5 8C Waiting Time is:2
6 82 Waiting Time is:5
7 84 Waiting Time is:2



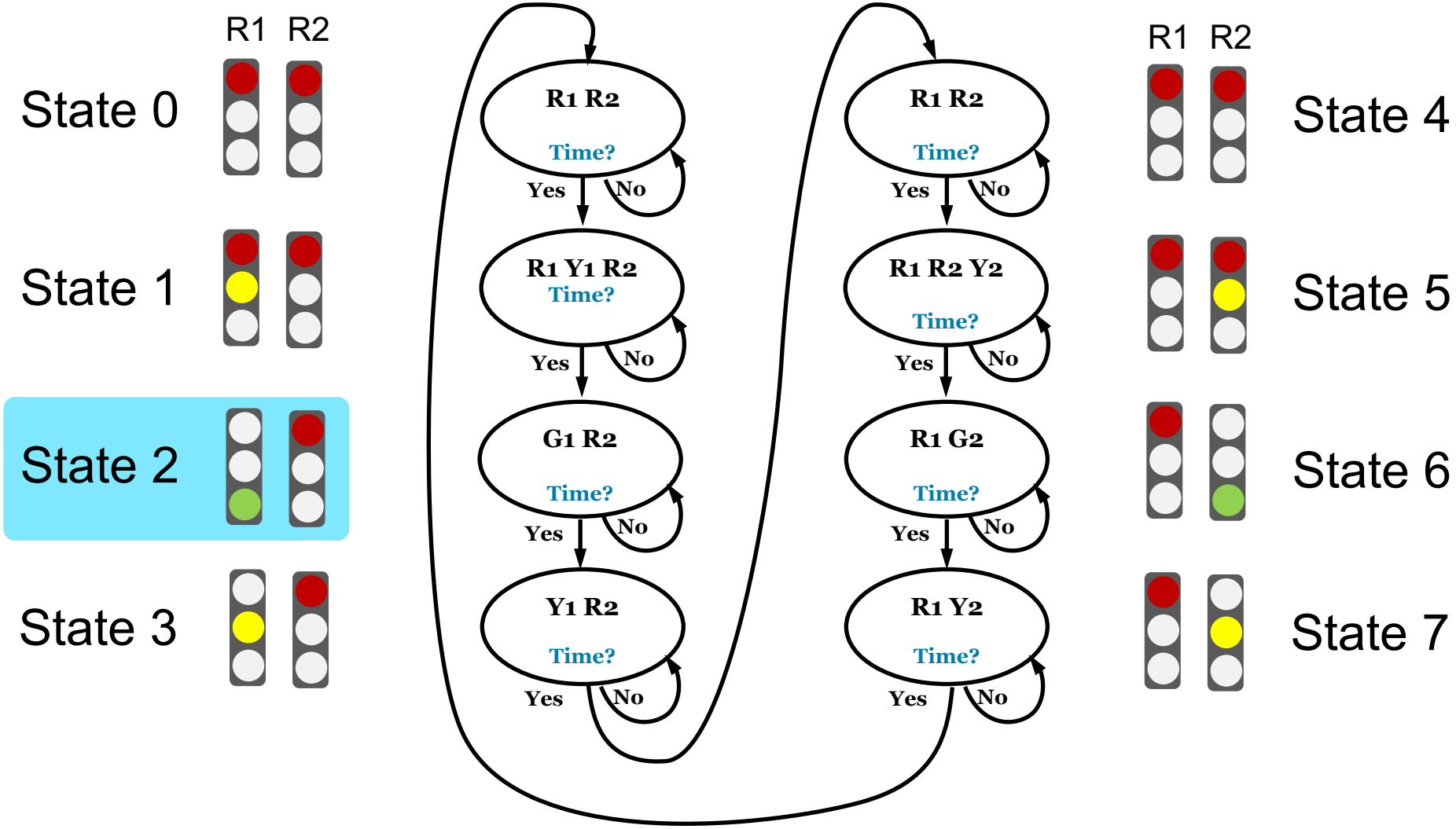
								Hex	Time
1	0	0	0	1	0	0	0	88	1
1	1	0	0	1	0	0	0	C8	2
0	0	1	0	1	0	0	0	28	5
0	1	0	0	1	0	0	0	48	2
1	0	0	0	1	0	0	0	88	1
1	0	0	0	1	1	0	0	8C	2
1	0	0	0	0	0	1	0	82	5
1	0	0	0	0	1	0	0	84	2

State Table Example – Traffic Lights

- This is fine for very simple situations
- Can be extended to cope with monitoring external inputs instead of just timing
  - For example, monitor port *B bit 1* which is connected to traffic sensor on *Road 2*
  - Have extra item in state table to indicate "*Road 1 stays on green until car detected on Road 2*"



State Table Example – Traffic Lights

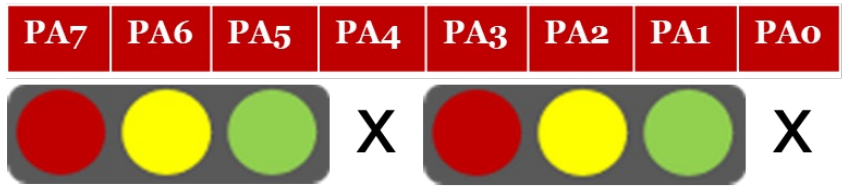


State Table Example – Traffic Lights

```
const int NumStates = 8;
int state = 0;
byte StateTable [NumStates][3]={
 {0x88, 1, 0},
 {0xC8, 2, 0},
 {0x28, 5, 1},
 {0x48, 2, 0},
 {0x88, 1, 0},
 {0x8C, 2, 0},
 {0x82, 5, 0},
 {0x84, 2, 0}};

void setup() {
 DDRA = 0xFF; // Set PORTA as output
Serial.begin(9600);
}

void loop() {
Serial.print(state);
Serial.print(' ');
PORTA = StateTable[state][0];
Serial.print(PORTA, HEX);
Serial.print(' ');
Serial.print("Waiting Time is:");
Serial.print(StateTable[state][1]);
Serial.print('\n');
delay(StateTable[state][1]*1000);

while(StateTable[state][3] && !(PINB & 0x01));
{
// Just wait here if StateTable[state][3] = 1 and Pin No 1 on Port B goes low (0).
}

if(state < NumStates-1)
{
state++;
}
else
{
  state = 0;
}}
```

Code output

```
0 88 Waiting Time is:1
1 C8 Waiting Time is:2
2 28 Waiting Time is:5
3 48 Waiting Time is:2
4 88 Waiting Time is:1
5 8C Waiting Time is:2
6 82 Waiting Time is:5
7 84 Waiting Time is:2
```

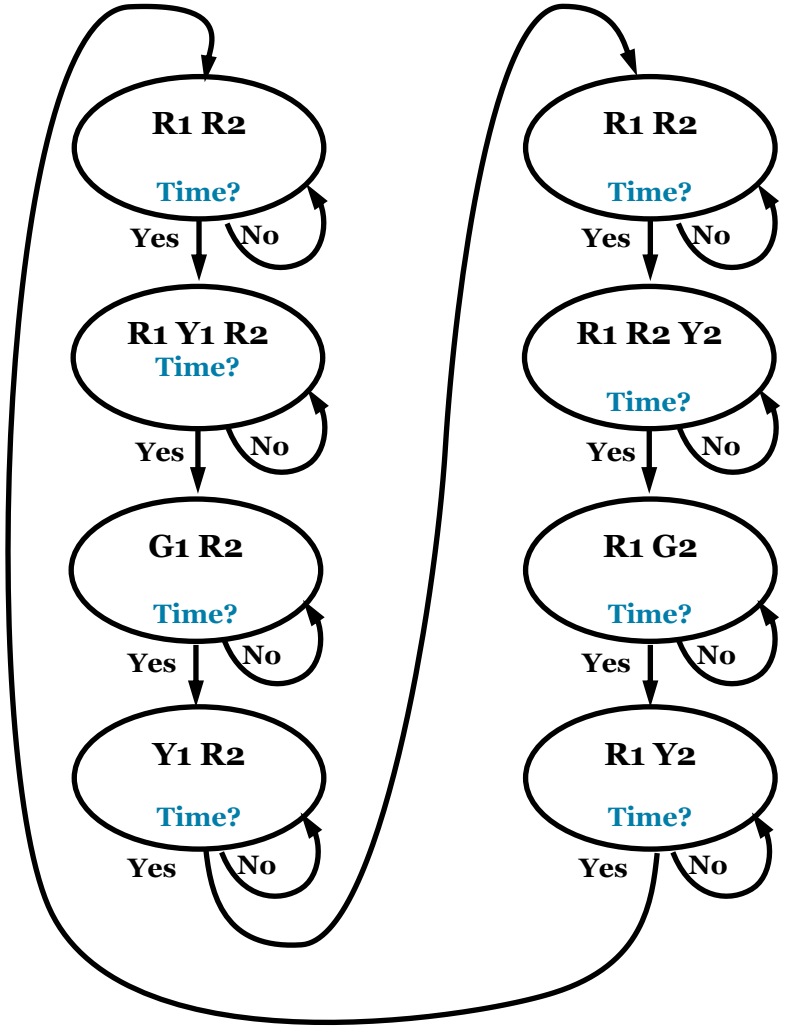


								Hex	Time
1	0	0	0	1	0	0	0	88	1
1	1	0	0	1	0	0	0	C8	2
0	0	1	0	1	0	0	0	28	5
0	1	0	0	1	0	0	0	48	2
1	0	0	0	1	0	0	0	88	1
1	0	0	0	1	1	0	0	8C	2
1	0	0	0	0	0	1	0	82	5
1	0	0	0	0	1	0	0	84	2

State Table Example – Traffic Lights

This is fine for simple situations

- In the examples so far, the sequence is always the same
- Conditions for moving to next stage may be simple or a little more complex
- But does not really cope with state diagrams *with branches*

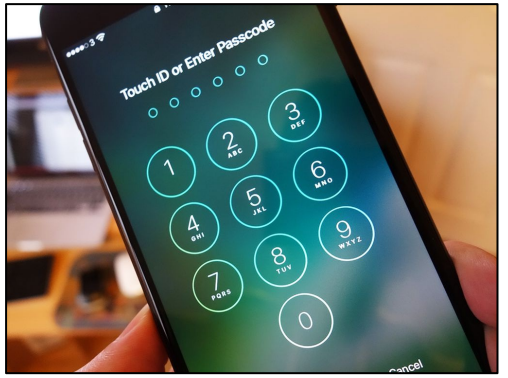


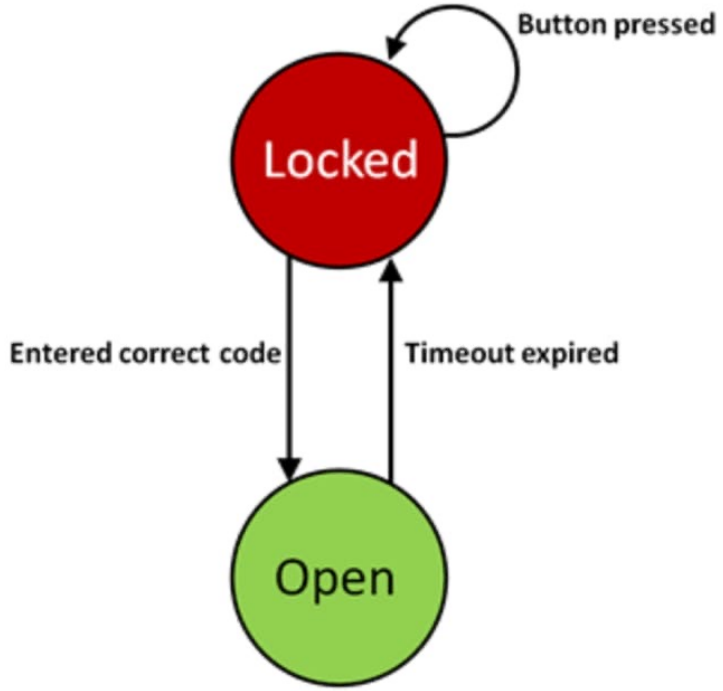
University of Nottingham
UK | CHINA | MALAYSIA

Finite State Machine

What is a Finite State Machine?

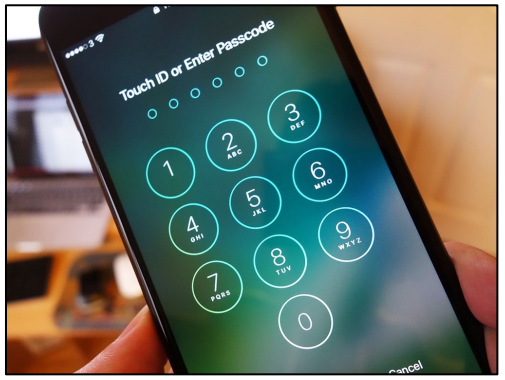
- A finite state machine (FSM) is a mathematical model of computation. It is an abstract machine that can be in a finite number of states. The machine can change from *one state to another* in response to some *inputs (conditions)*; the change from one state to another is called a *transition*.
- FSMs are used to model a wide variety of systems, including:
  - Digital circuits
  - Software controllers
  - Networking protocols
  - Natural language processing
  - Video games
- The FSM is a powerful way to design a program with
  - Events
  - Sequences
- Also used in menus for interactive programs

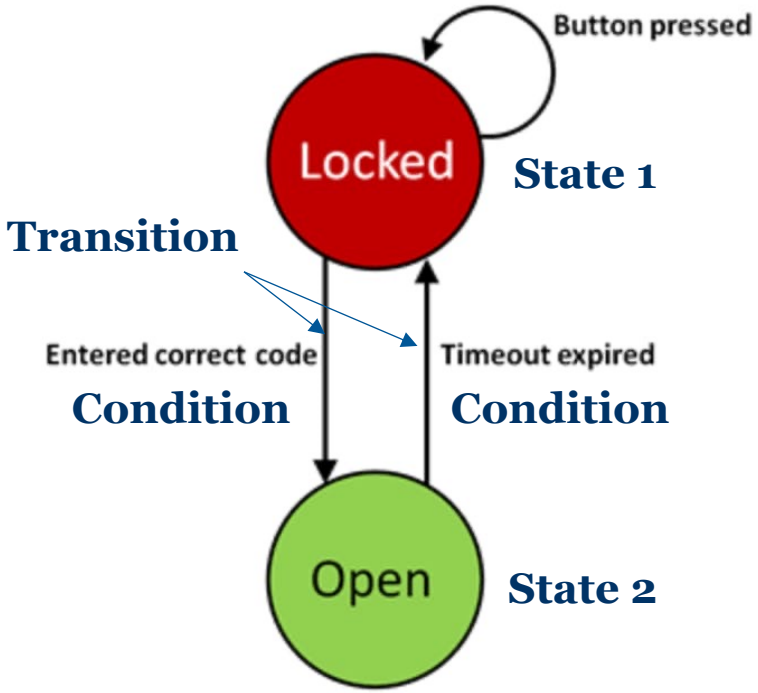




What is a Finite State Machine?

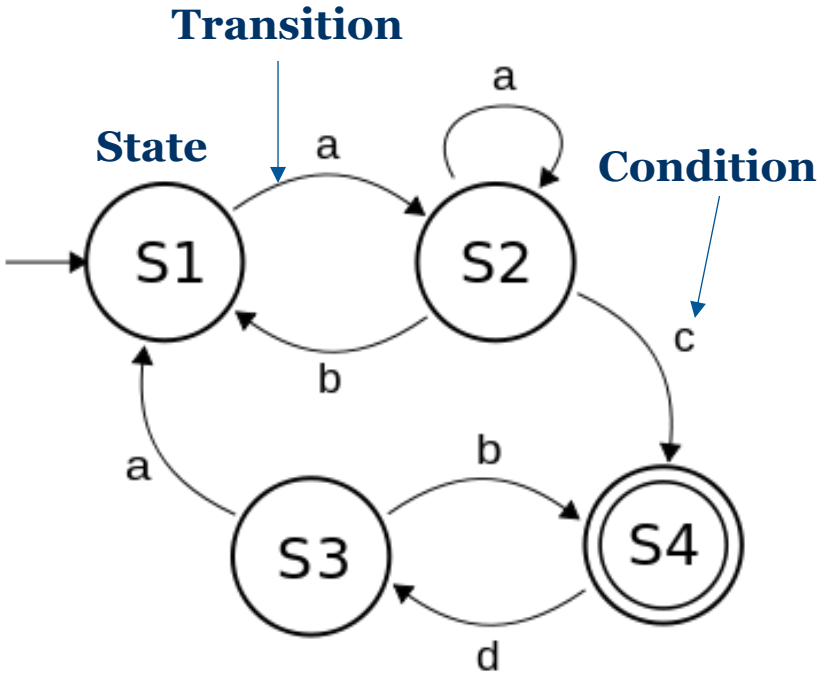
- In our Finite State Machine program, we consider:
  - **States**: typically, a stage of a process involving some functionality, something your "machine" will carry on doing until some condition is satisfied
  - **Condition**: some criterion that is satisfied e.g., an external input taking some value resulting in the machine changing state
  - **Transition**: the act of changing to a new state





What is a Finite State Machine?

- In our Finite State Machine program, we consider:
  - **States**: typically, a stage of a process involving some functionality, something your "machine" will carry on doing until some condition is satisfied
  - **Condition**: some criterion that is satisfied e.g., an external input taking some value resulting in the machine changing state
  - **Transition**: the act of changing to a new state



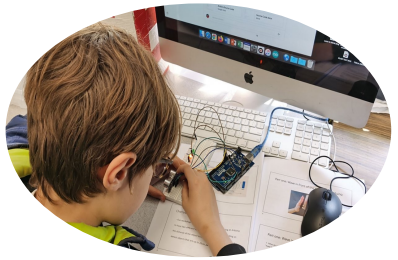
An FSM Real-Life Example

- A student may take various **states**:
  - Working
  - In room and bored
  - Socialising with a friend
- We also consider the *default state* of "At home with parents"
- We ignore the cases of sleeping, going out and eating – this student has spent all his money on a laptop and an Arduino to learn Mechatronics and is too interested in learning it to sleep ☺

An FSM Real-Life Example

The following events (along with current state) will influence student's actions (**conditions**):

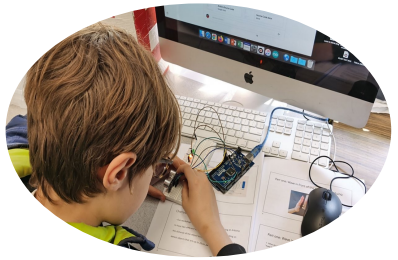
- Work needs doing
- Work finished
- Friend calls
- "Friend" gets obnoxious
- Start of term
- End of term



An FSM Real-Life Example

The actions corresponding to the **transitions** are:

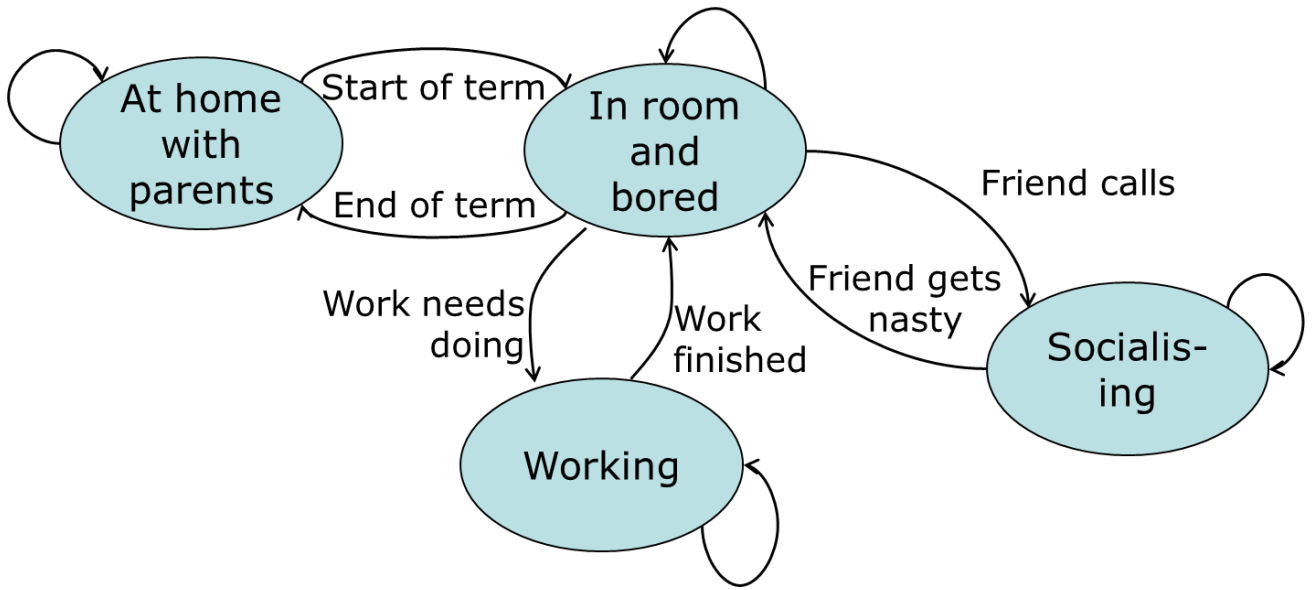
- Make a start on work
- Put work away
- Start chatting to friend
- Kick “friend” out of room
- Travel to university
- Travel home



An FSM Real-Life Example

• In room & bored	+ Work needs doing	→ Make a start on work
• Working	+ Work finished	→ Put work away
• In room & bored	+ Friend calls	→ Start chatting to friend
• Socialising	+ "Friend" gets obnoxious	→ Kick "friend" out of room
• At home	+ Start of term	→ Travel to university
• In room & bored	+ End of term	→ Travel home

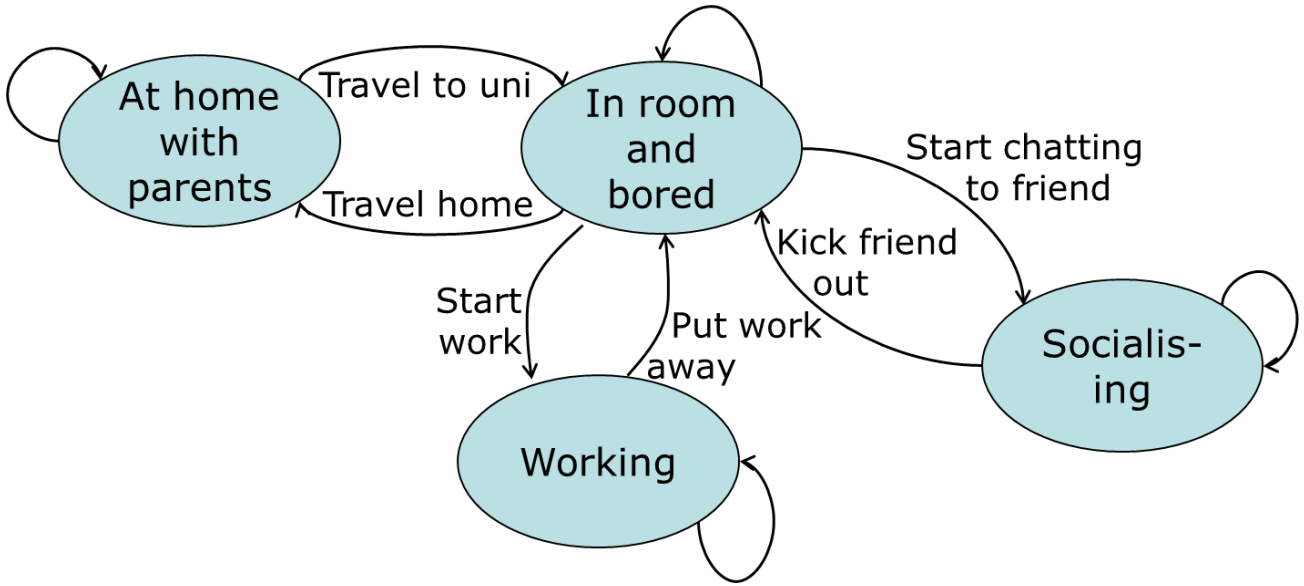
We can represent this via a "*state transition diagram*", showing the *events (conditions)* prompting each transition.



An FSM Real-Life Example

• In room & bored	+ Work needs doing	→ Make a start on work
• Working	+ Work finished	→ Put work away
• In room & bored	+ Friend calls	→ Start chatting to friend
• Socialising	+ "Friend" gets obnoxious	→ Kick "friend" out of room
• At home	+ Start of term	→ Travel to university
• In room & bored	+ End of term	→ Travel home

Similarly, we can show the *actions (transitions)* involved in each transition



An FSM Real-Life Example

- In room & bored + Work needs doing → Make a start on work
- Working + Work finished → Put work away
- In room & bored + Friend calls → Start chatting to friend
- Socialising + "Friend" gets obnoxious → Kick "friend" out of room
- At home + Start of term → Travel to university
- In room & bored + End of term → Travel home

Can now draw table showing **actions (transitions)** for each combination of current state and event (condition); greyed-out combinations have no action:

	At home	In room	Working	Socialising
Term starts	Travel to uni			
Term ends		Travel home		
Work needed		Start work		
Work finished			Put away	
Friend calls		Start chatting		
Friend nasty				Kick friend out

An FSM Real-Life Example

• In room & bored	+ Work needs doing	→ Make a start on work
• Working	+ Work finished	→ Put work away
• In room & bored	+ Friend calls	→ Start chatting to friend
• Socialising	+ "Friend" gets obnoxious	→ Kick "friend" out of room
• At home	+ Start of term	→ Travel to university
• In room & bored	+ End of term	→ Travel home

We can draw another table giving the **new states** (sometimes amalgamated with first table by having two fields in each box in table)

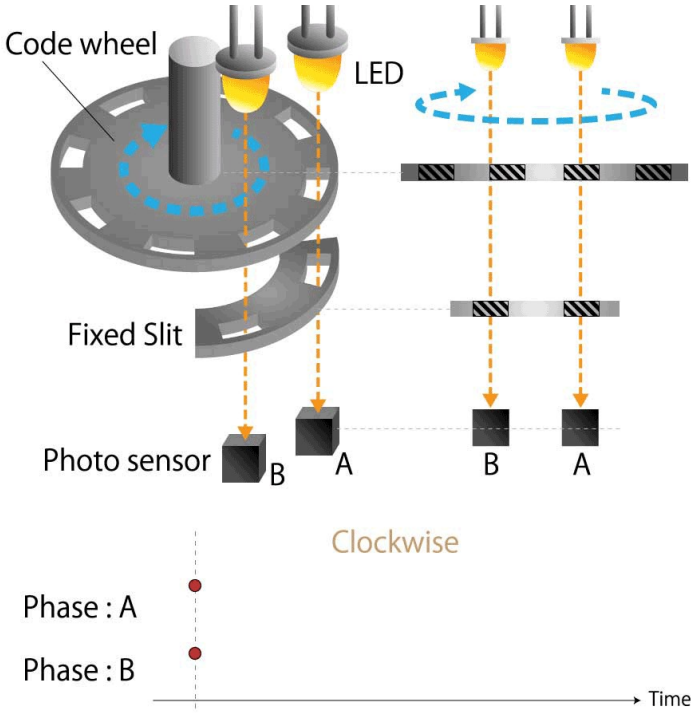
	At home	In room	Working	Socialising
Term starts	In room			
Term ends		At home		
Work needed		Working		
Work finished			In room	
Friend calls		Socialising		
Friend nasty				In room

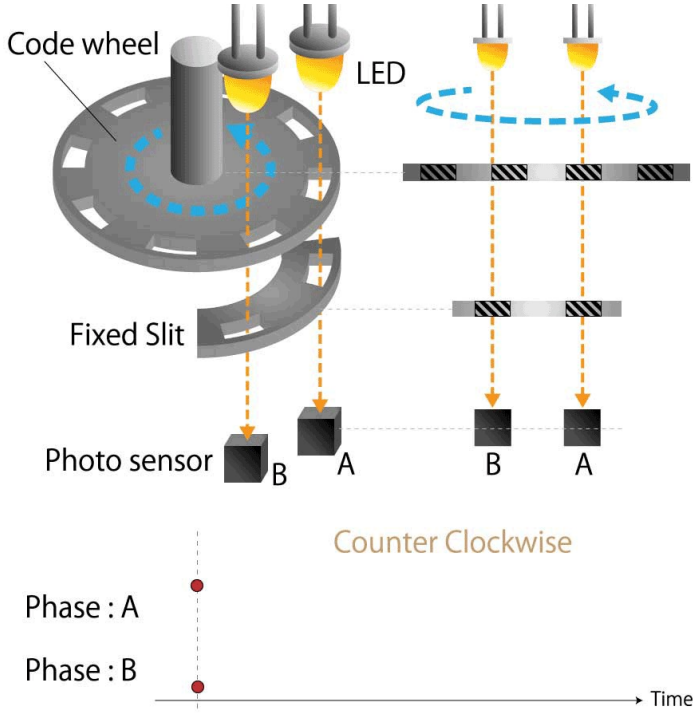
University of Nottingham
UK | CHINA | MALAYSIA

FSM
Practical Example

FSM – Practical Example

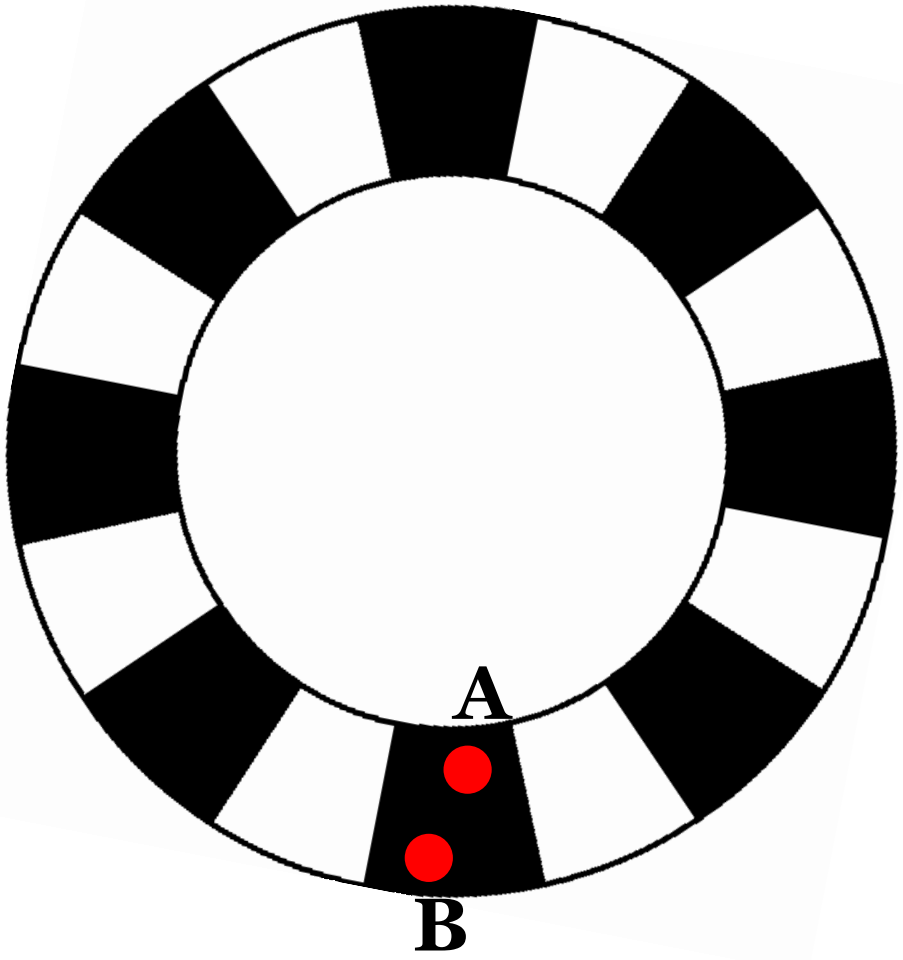
- Consider an example which ties in nicely with one of the labs (Lab 1)
- Consider the example of a *rotary encoder*: used for measuring position of a leadscrew or similar on a machine tool
- Disc with slits attached to rotating screw and two stationary photo-sensors
- As disc rotates, photo-sensors are activated and de-activated

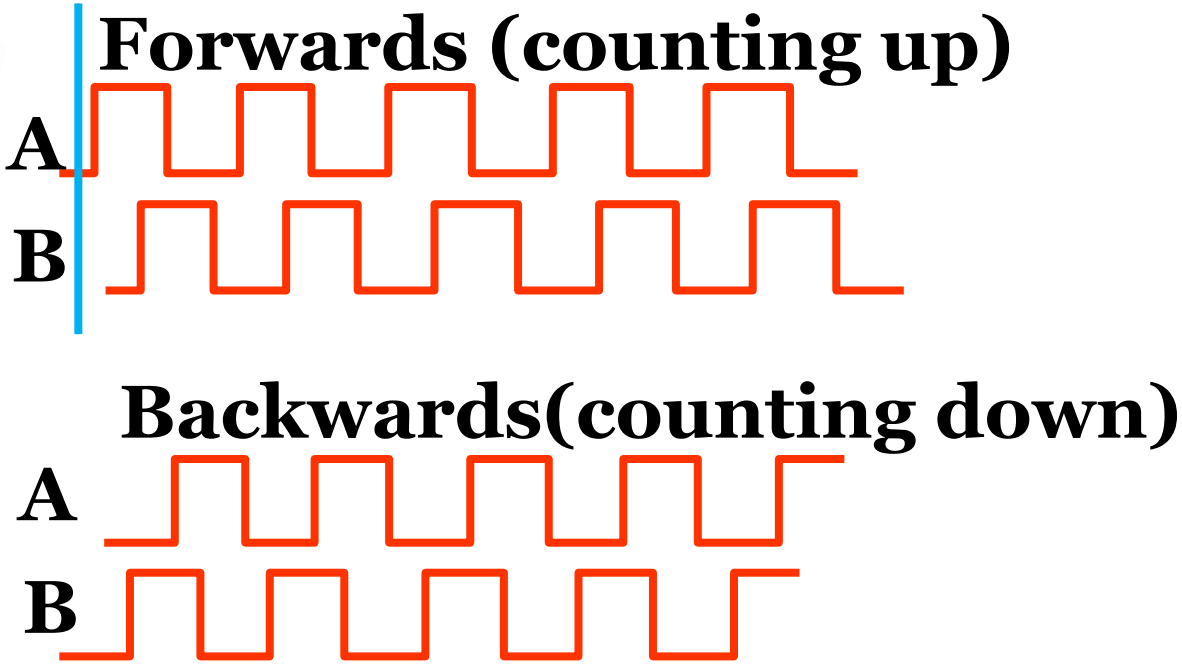




FSM – Practical Example

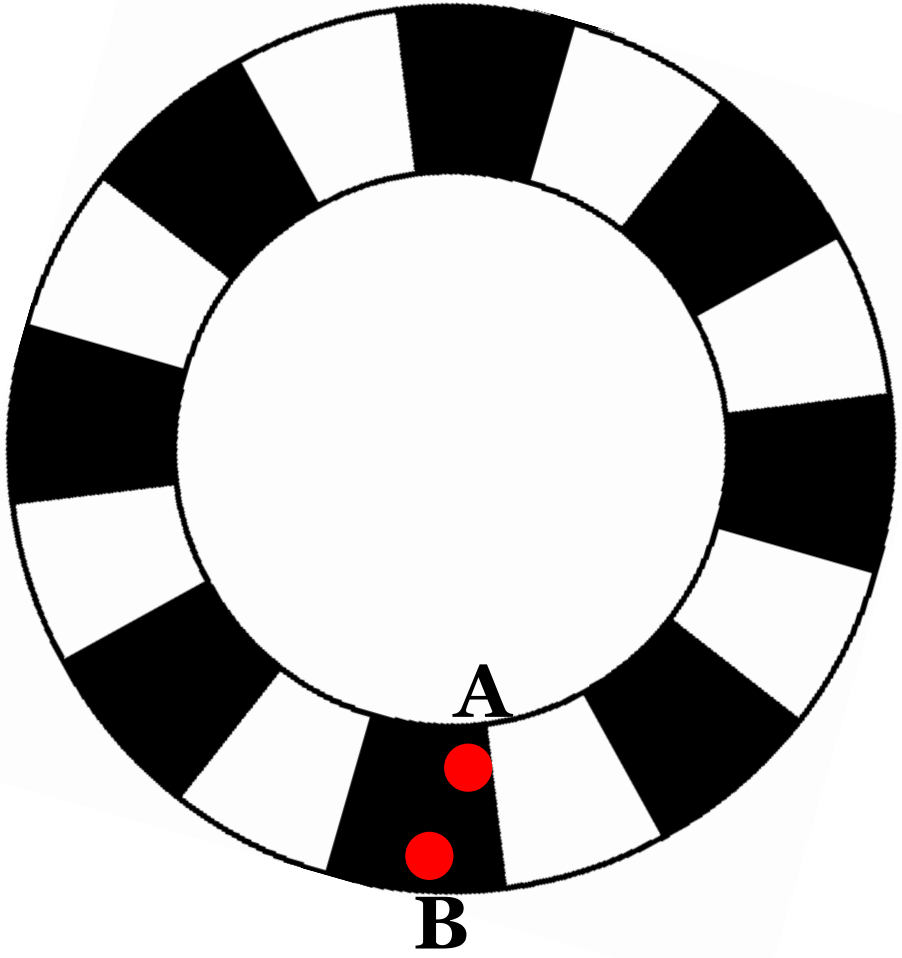
Rotary Incremental Encoder

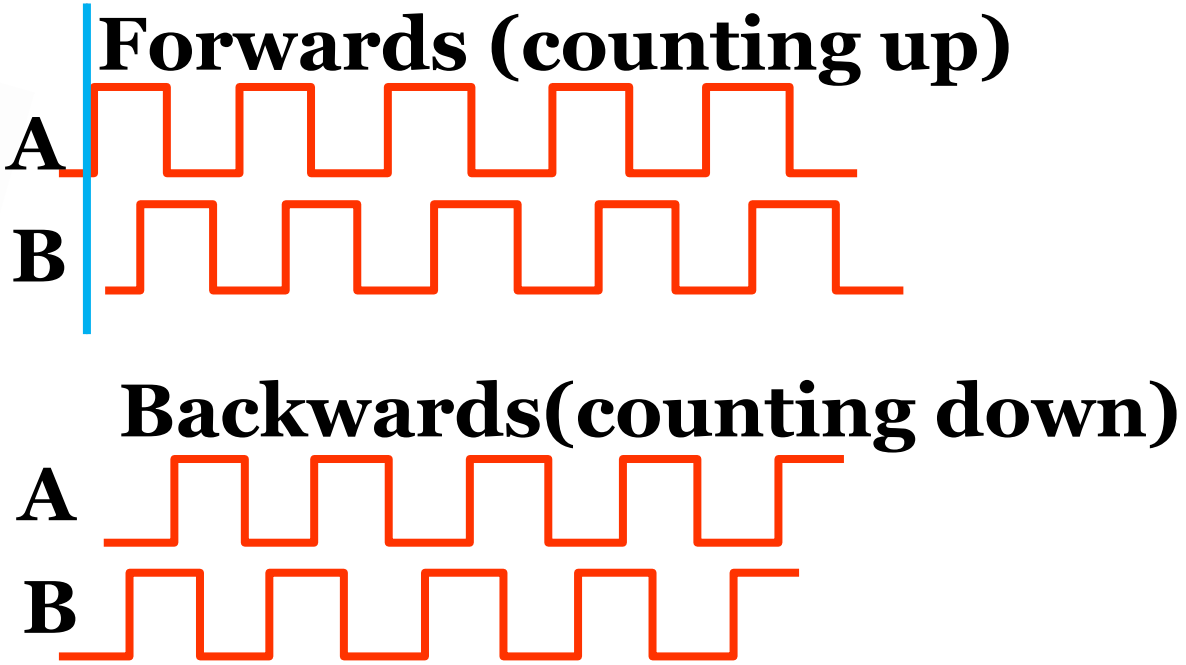




FSM – Practical Example

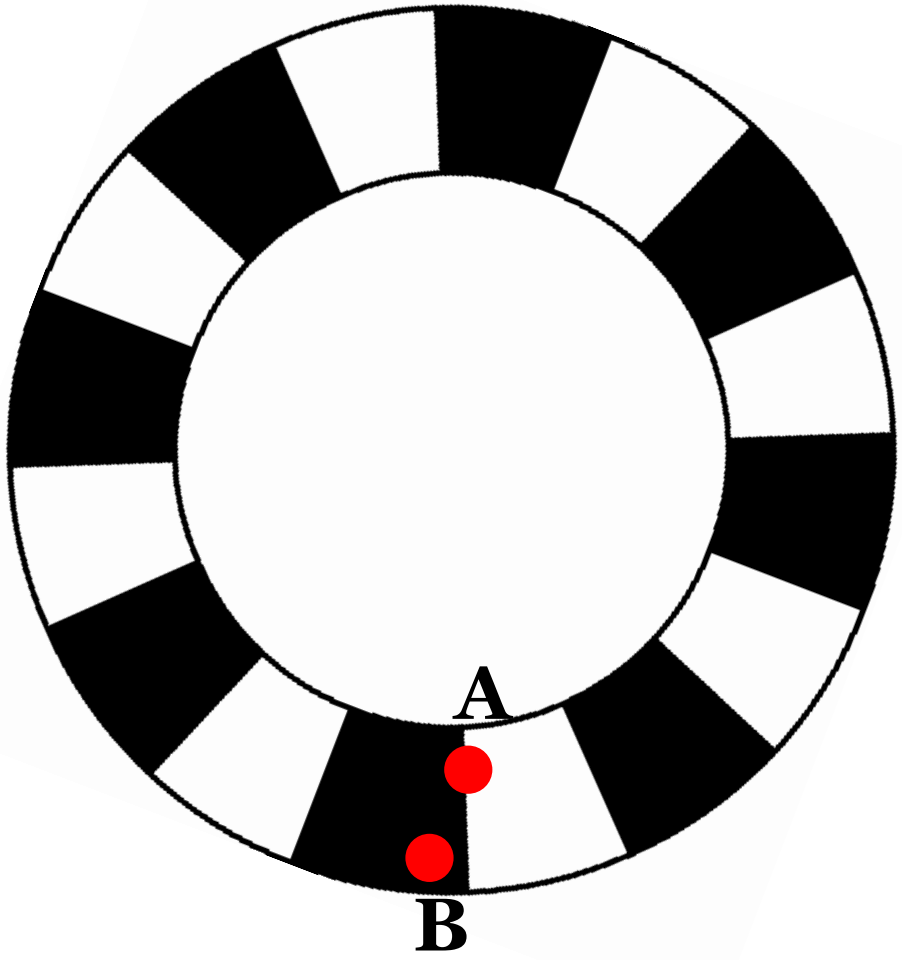
Rotary Incremental Encoder

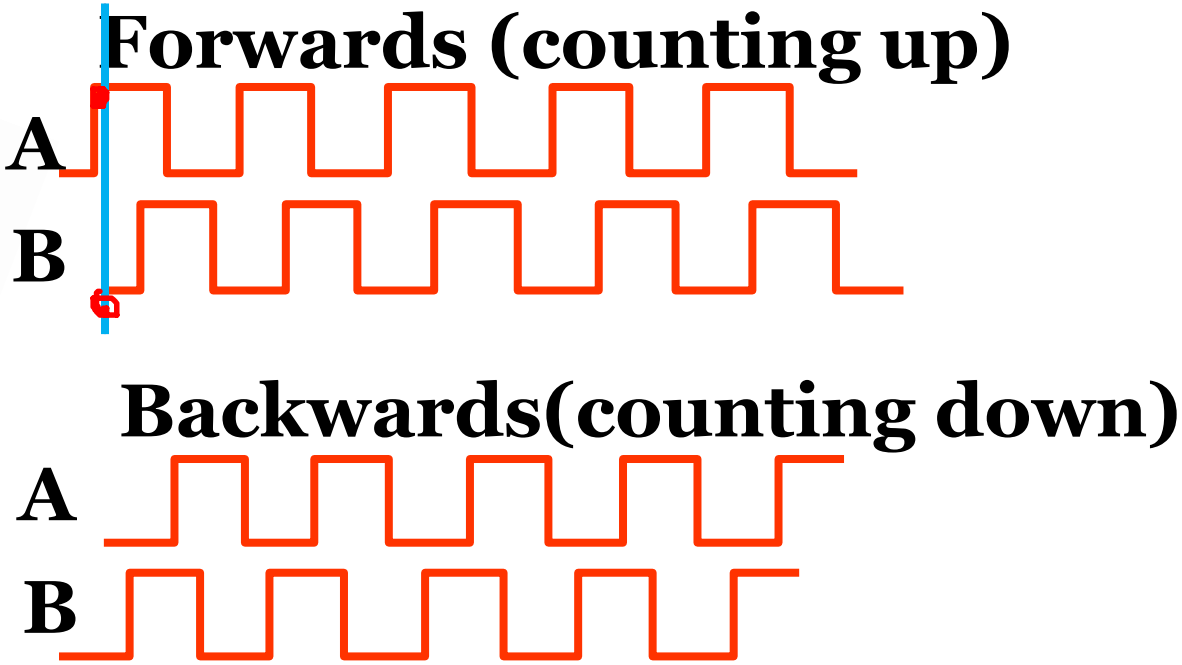




FSM – Practical Example

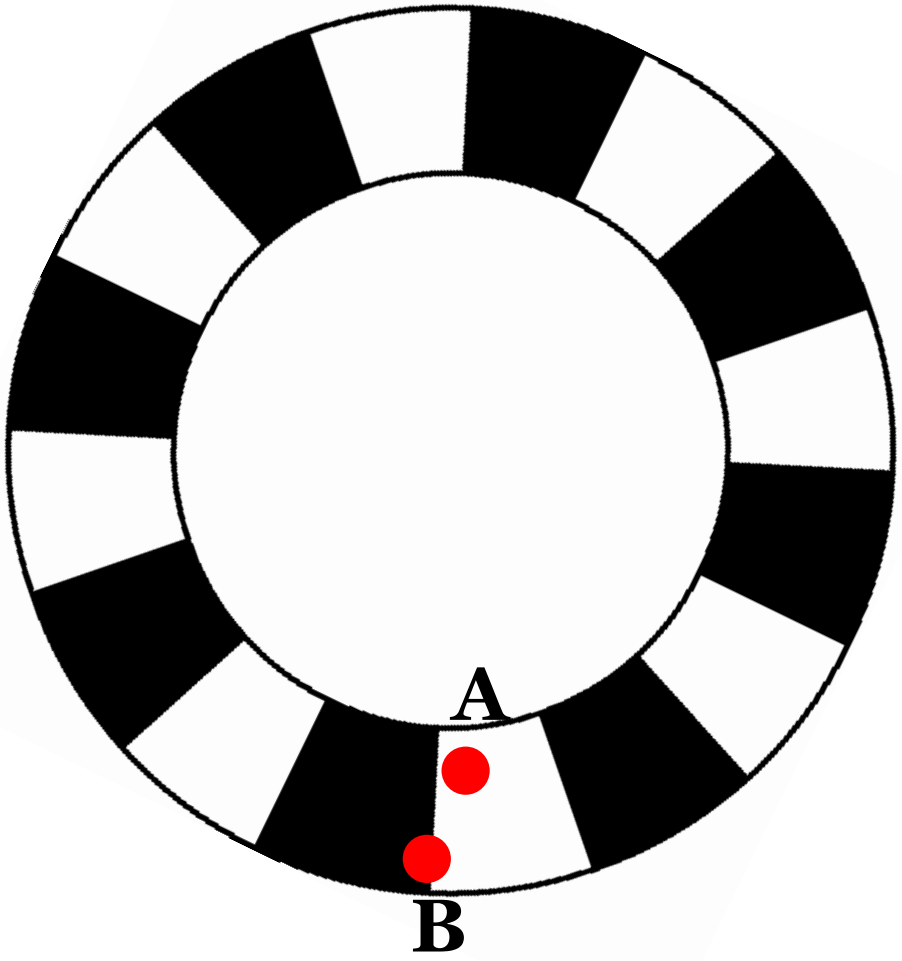
Rotary Incremental Encoder

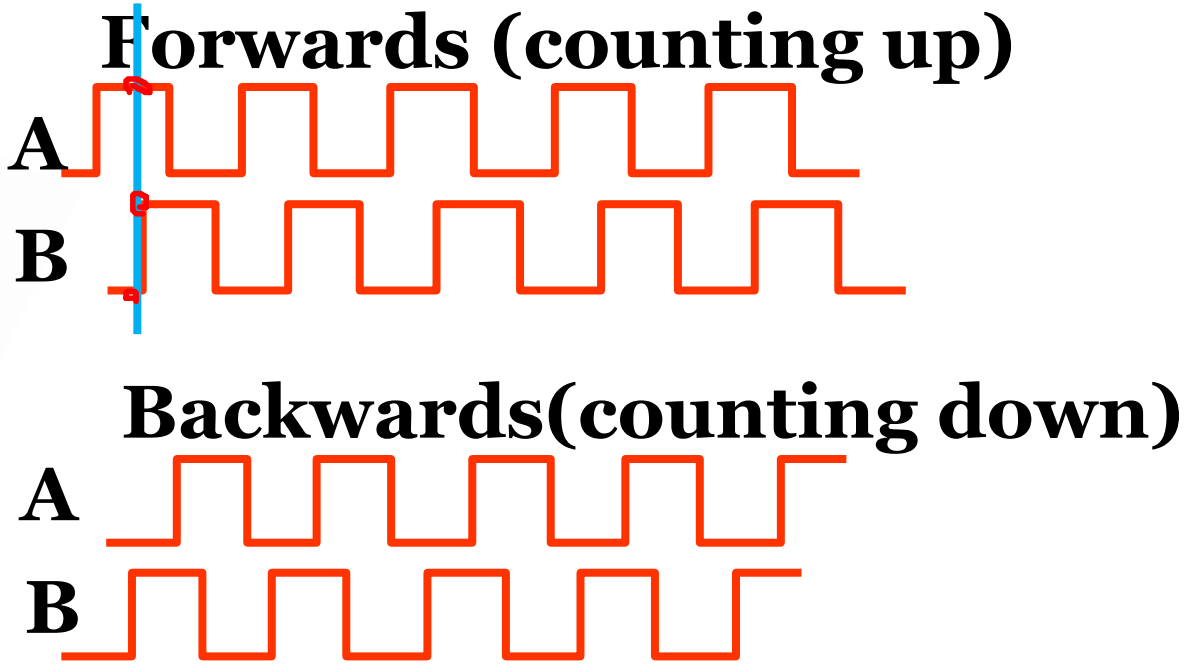




FSM – Practical Example

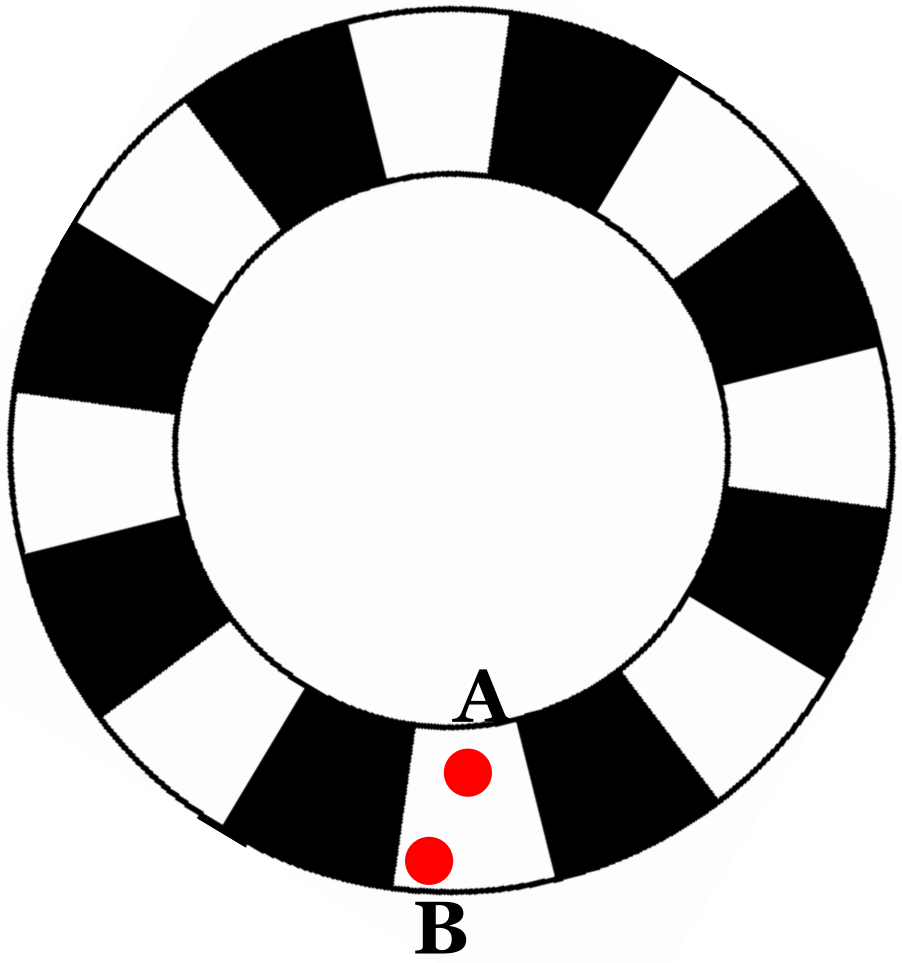
Rotary Incremental Encoder

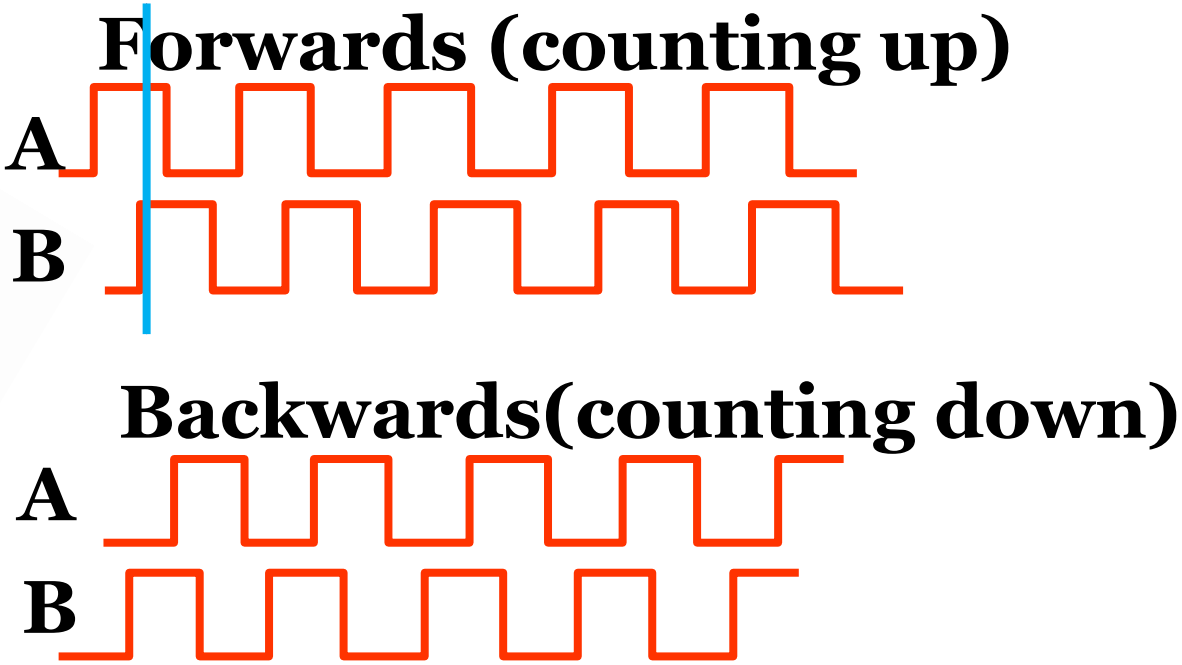




FSM – Practical Example

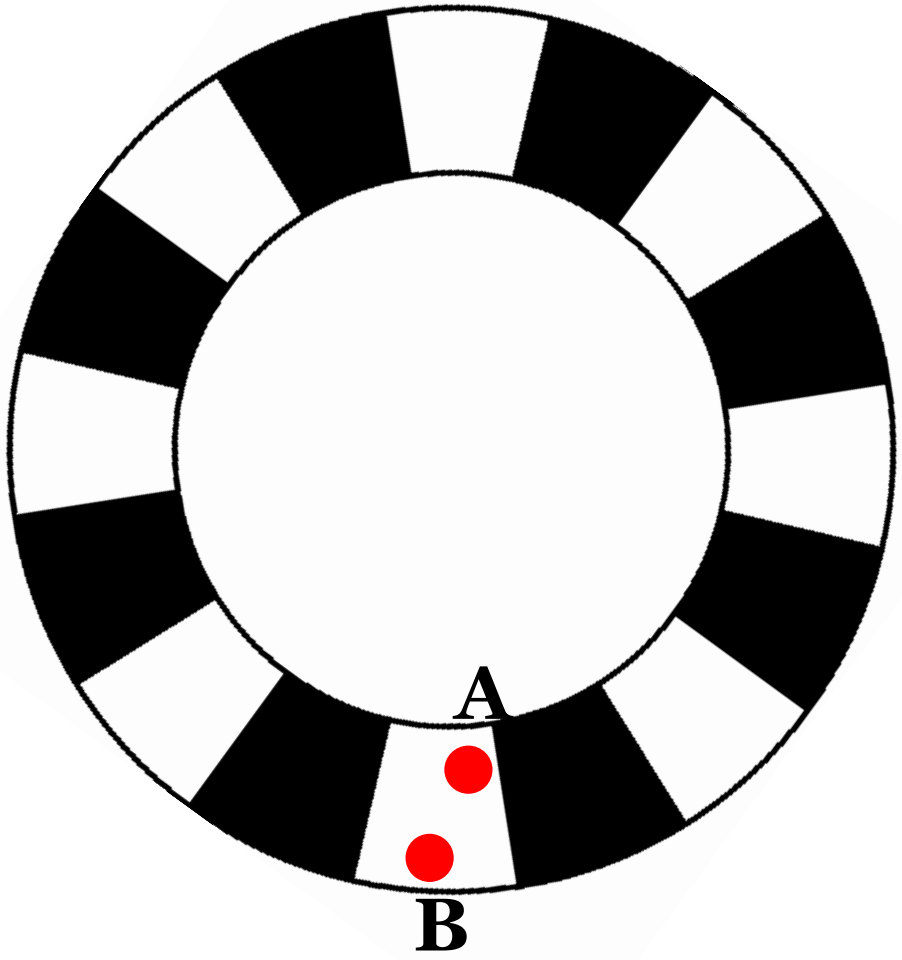
Rotary Incremental Encoder

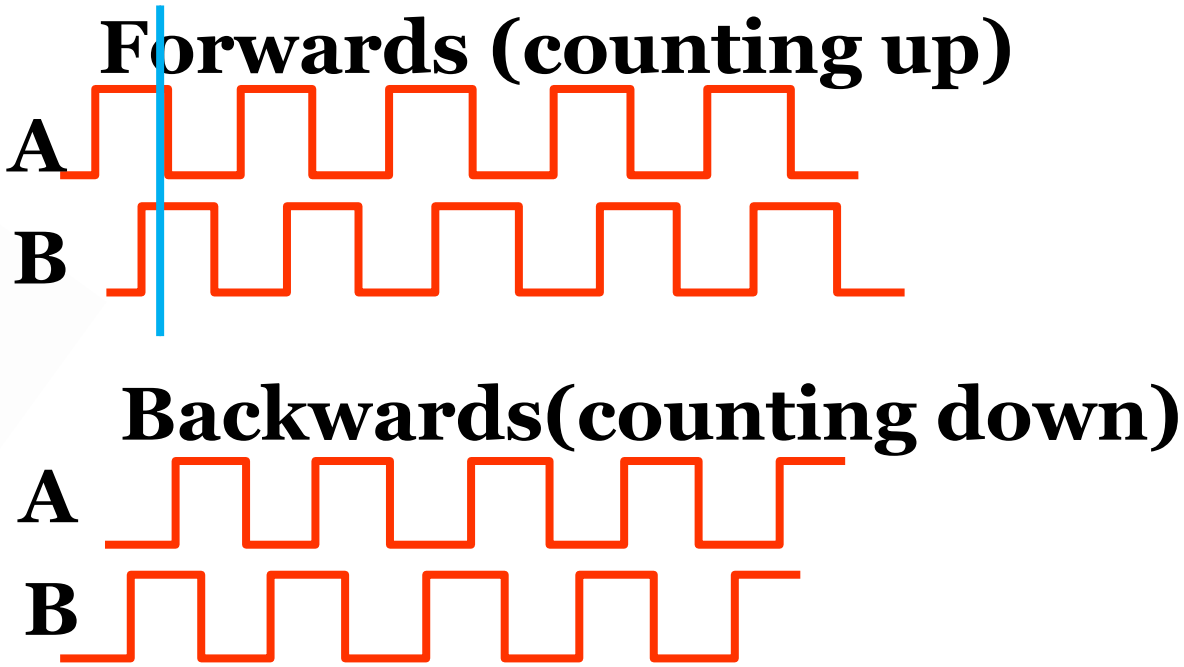




FSM – Practical Example

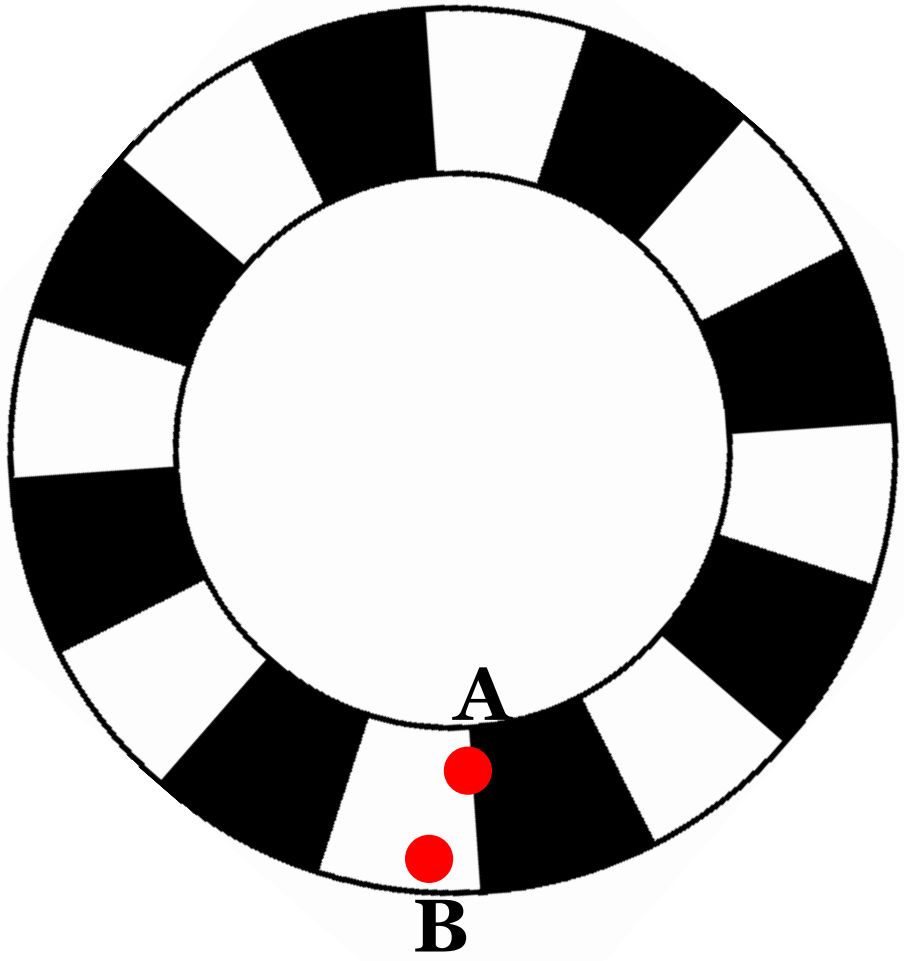
Rotary Incremental Encoder

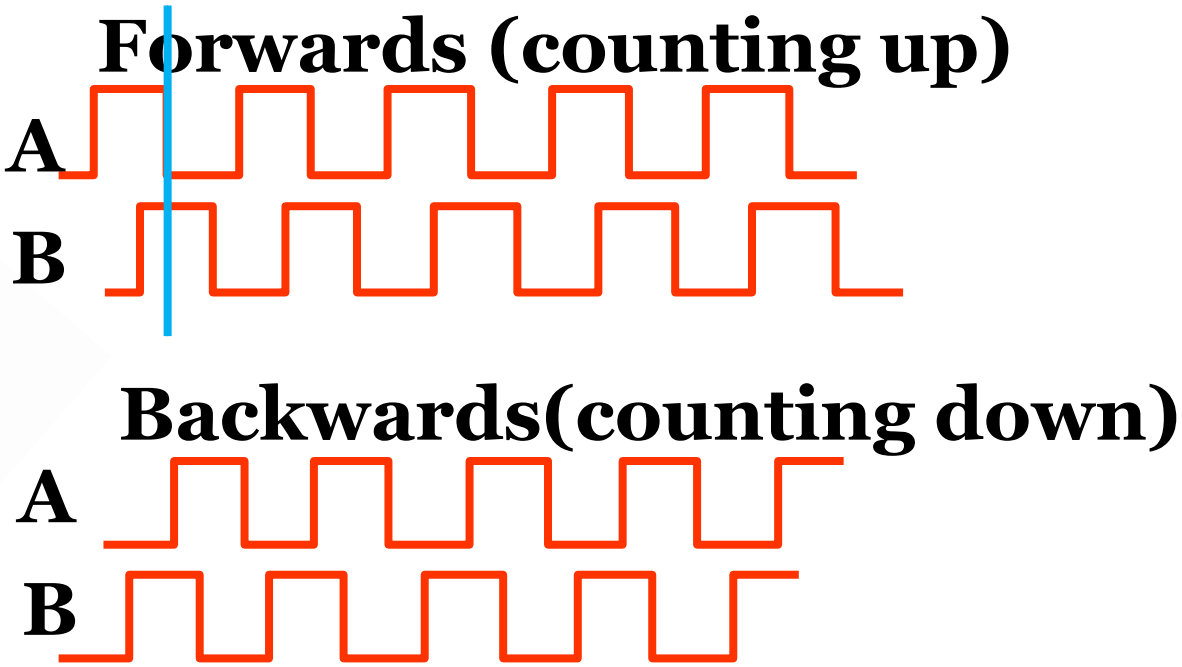




FSM – Practical Example

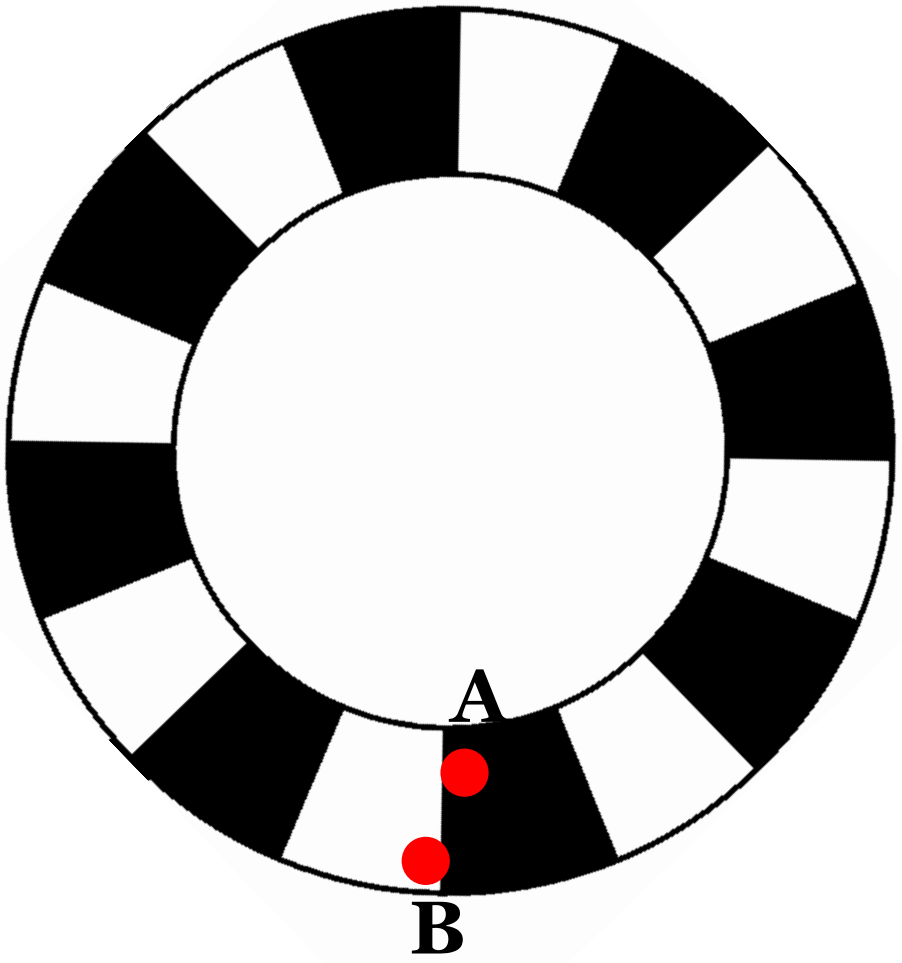
Rotary Incremental Encoder

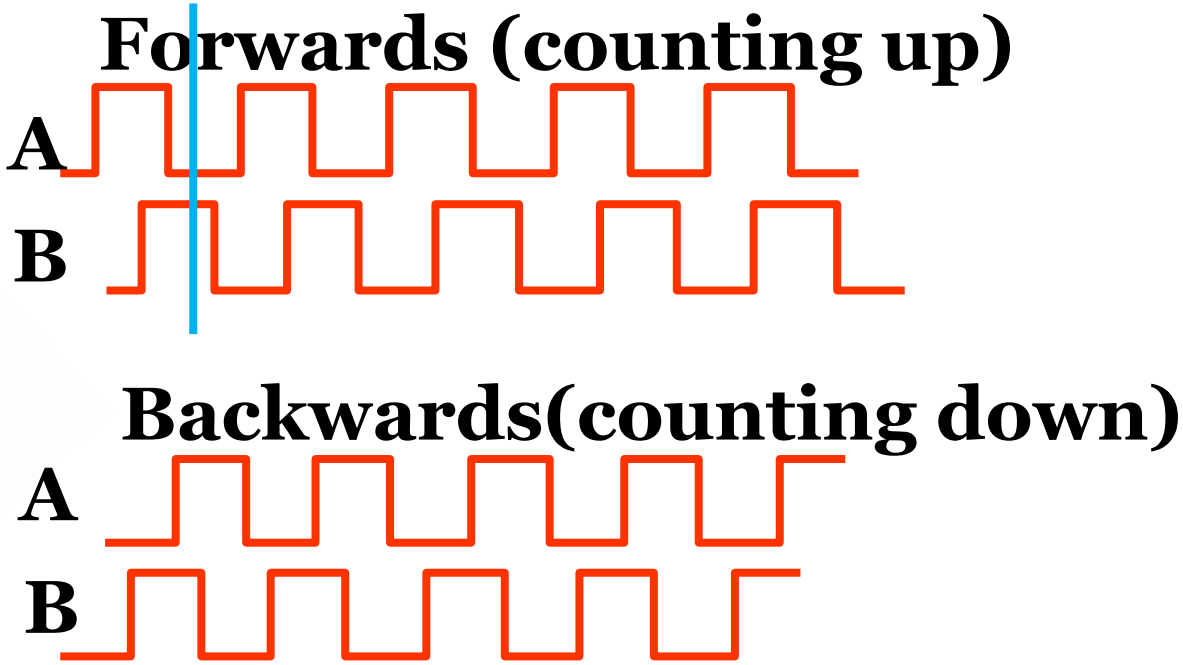




FSM – Practical Example

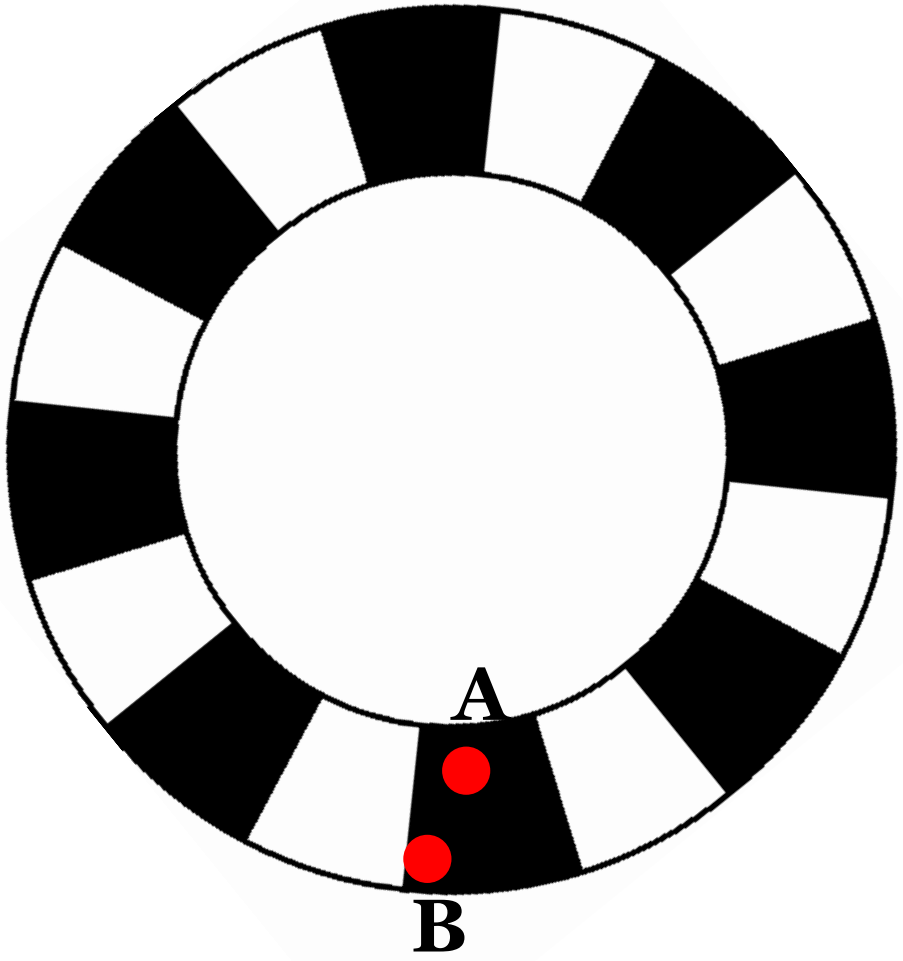
Rotary Incremental Encoder

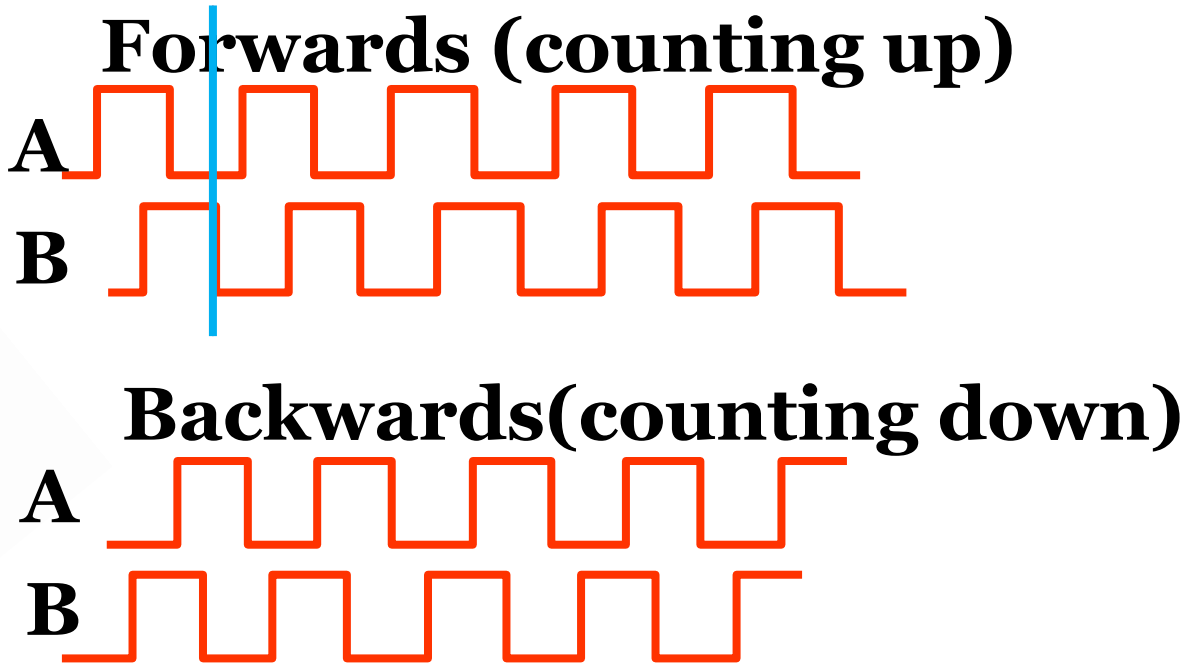




FSM – Practical Example

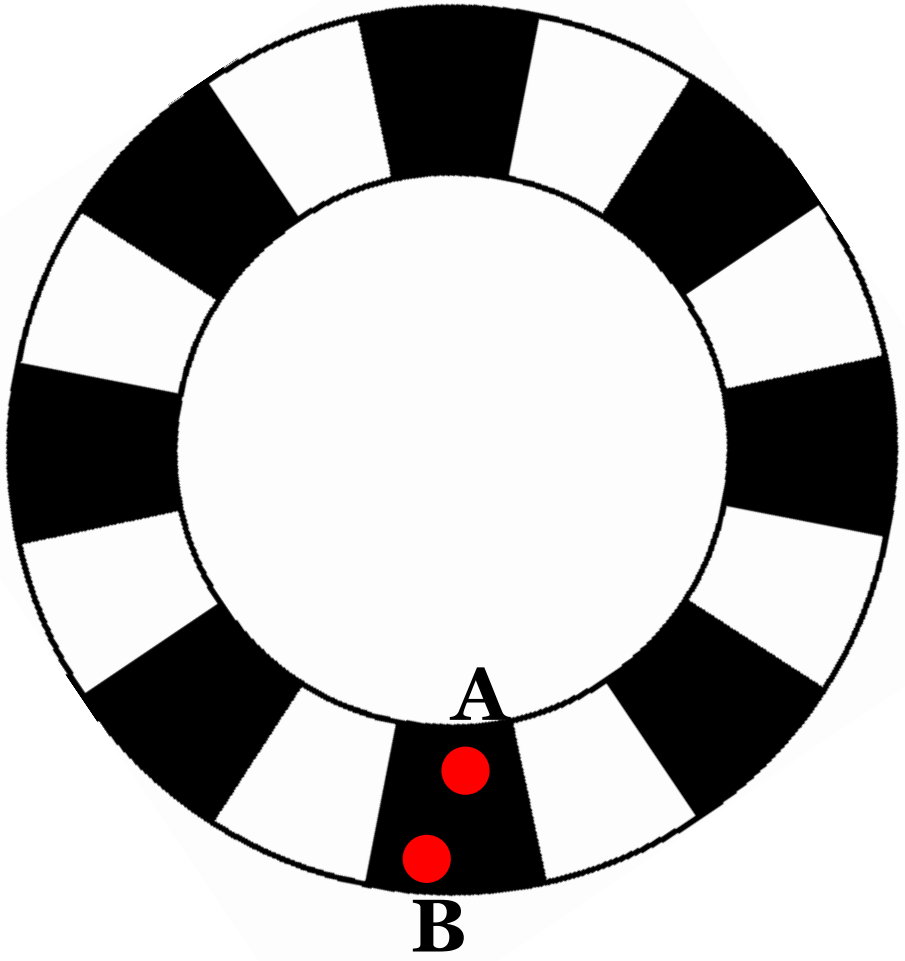
Rotary Incremental Encoder

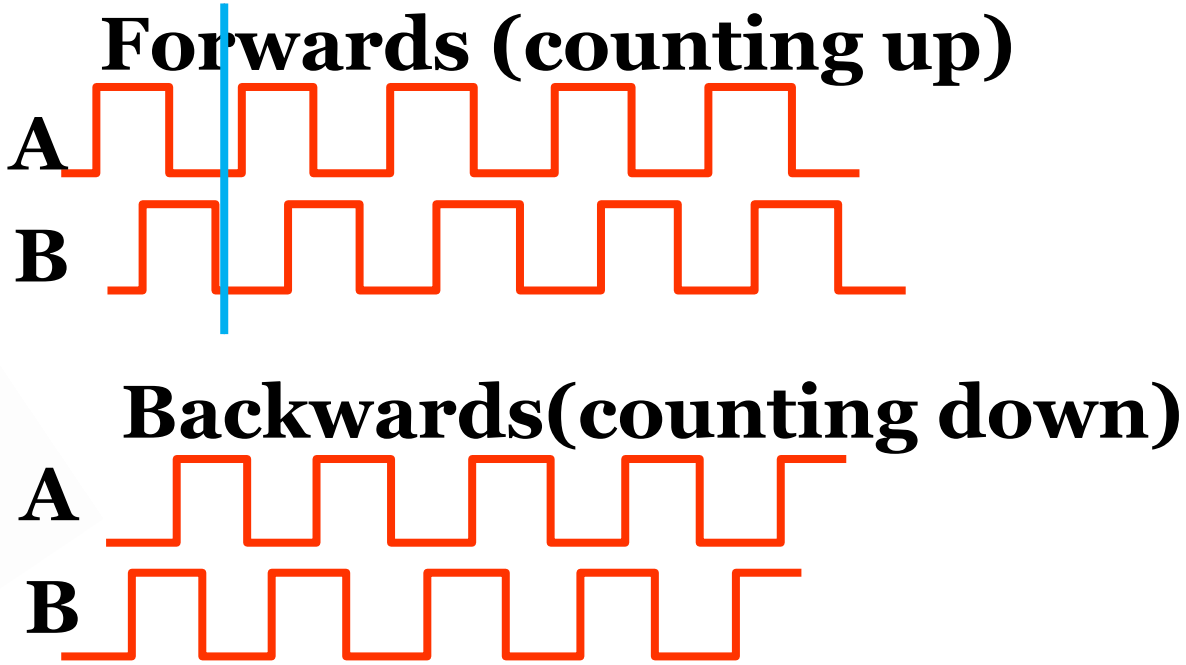




FSM – Practical Example

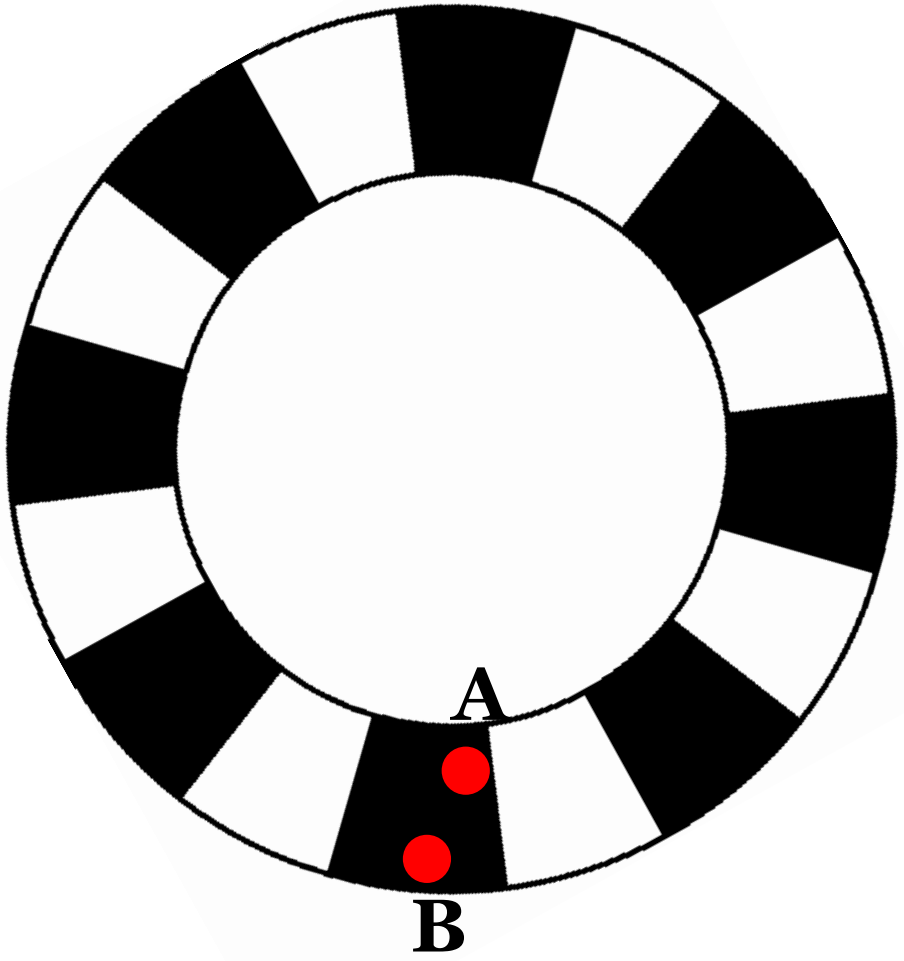
Rotary Incremental Encoder

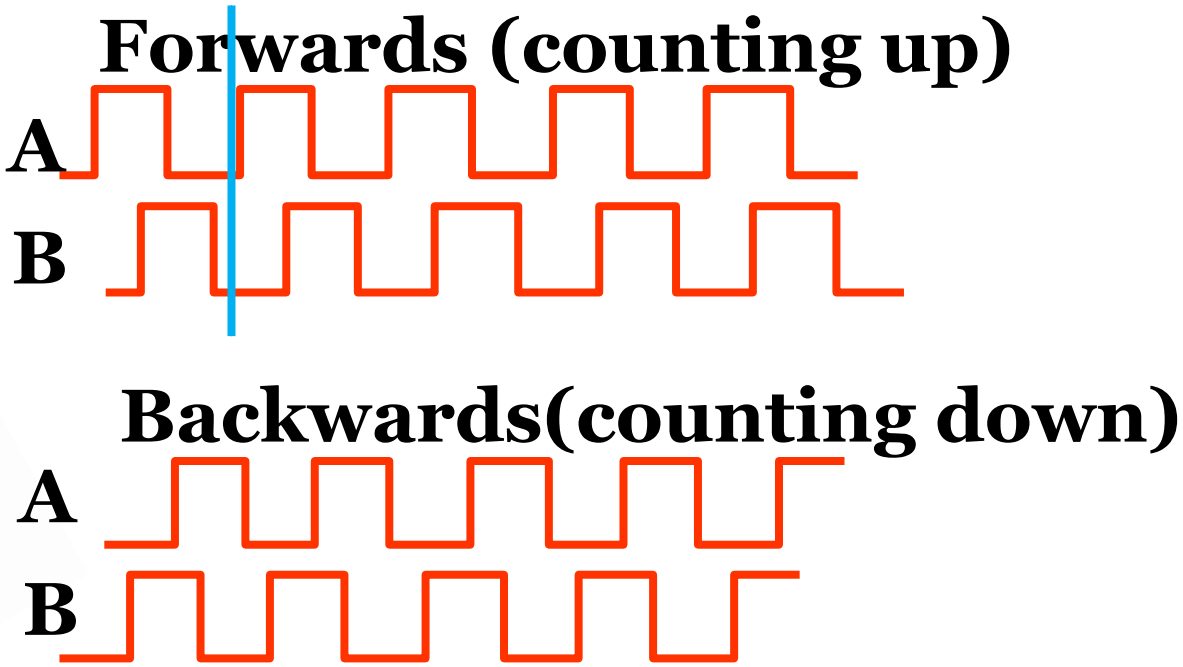




FSM – Practical Example

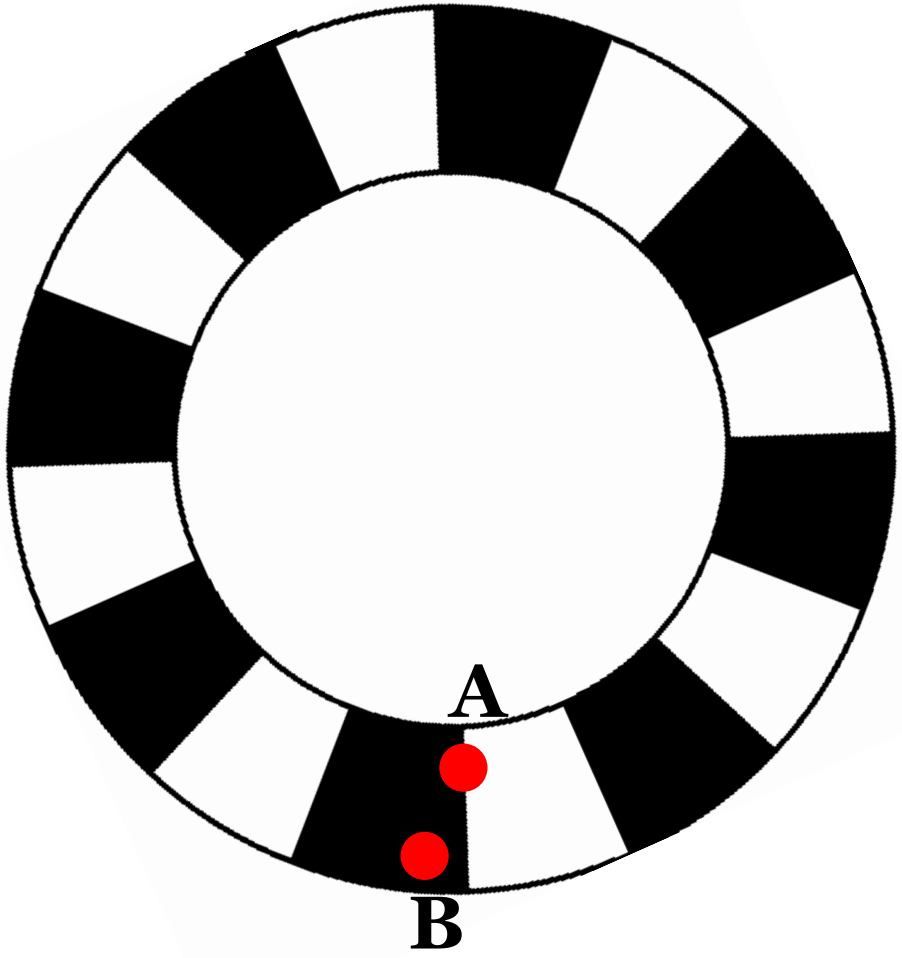
Rotary Incremental Encoder

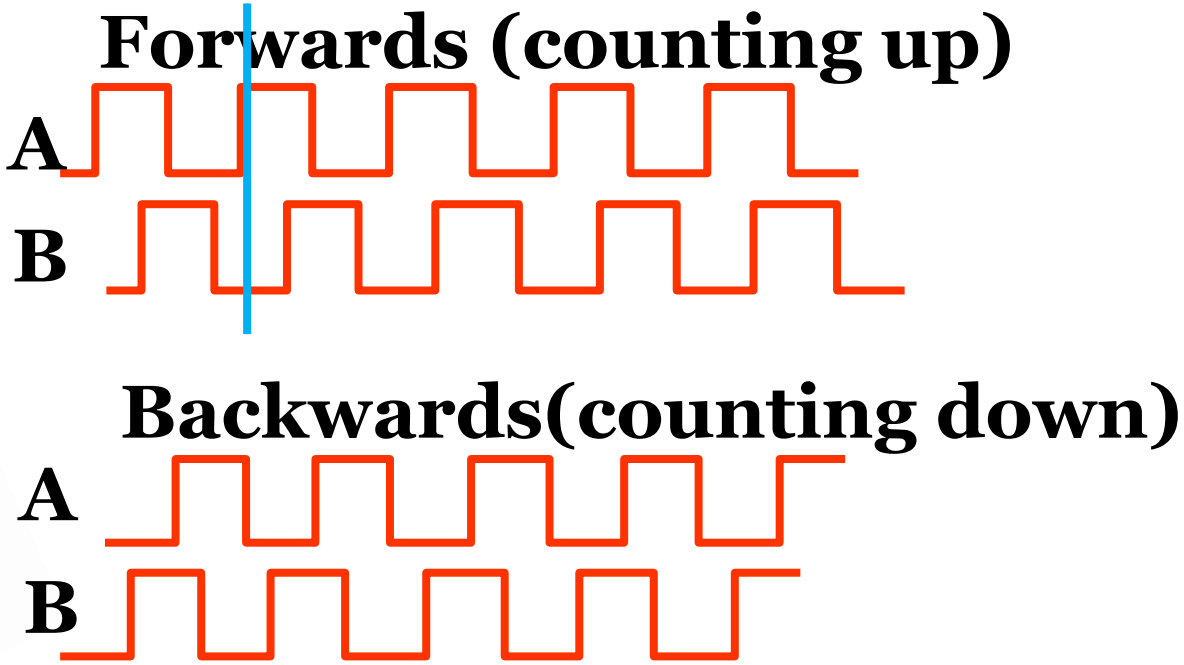




FSM – Practical Example

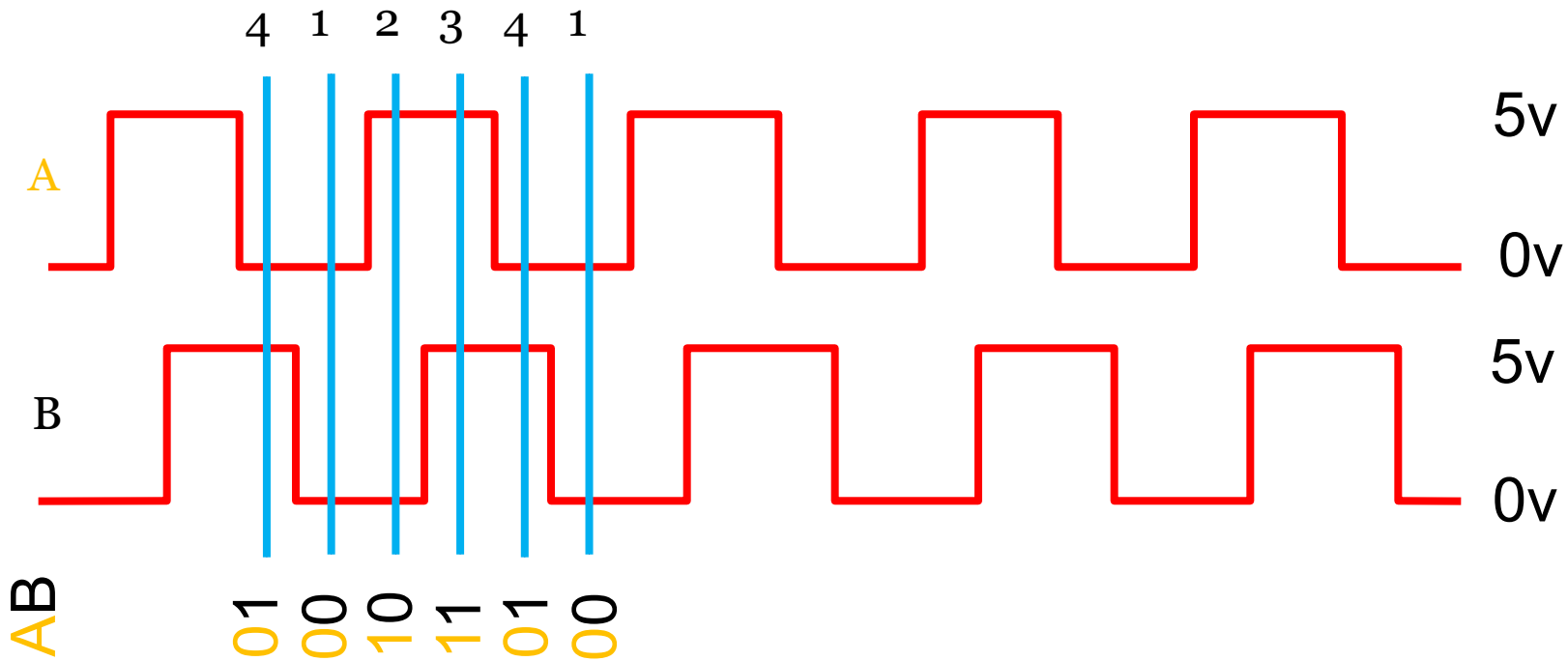
Rotary Incremental Encoder





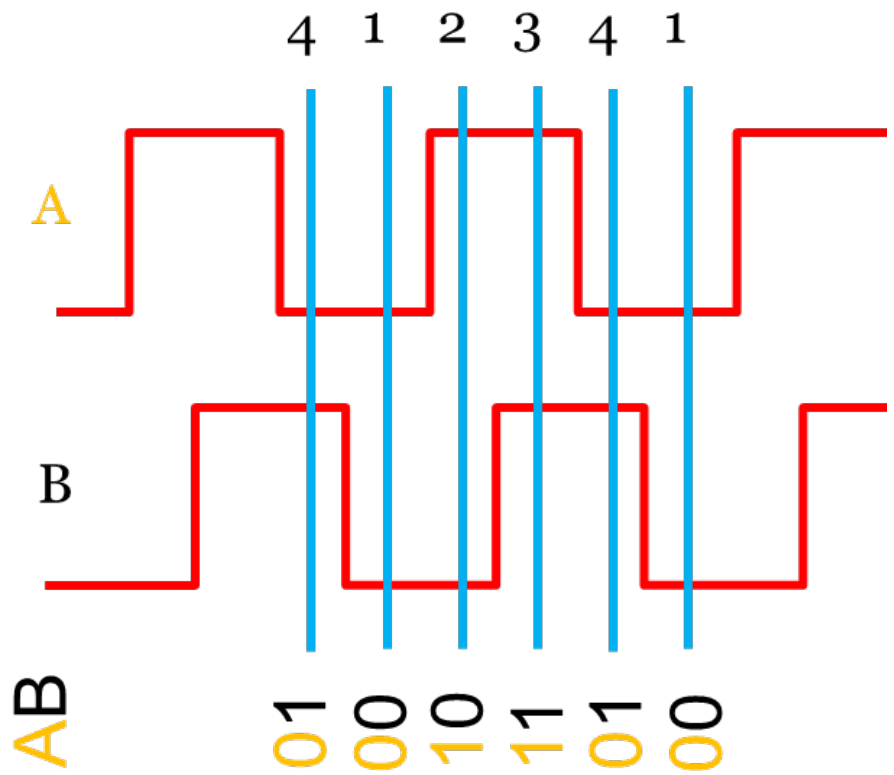
Rotary Incremental Encoder

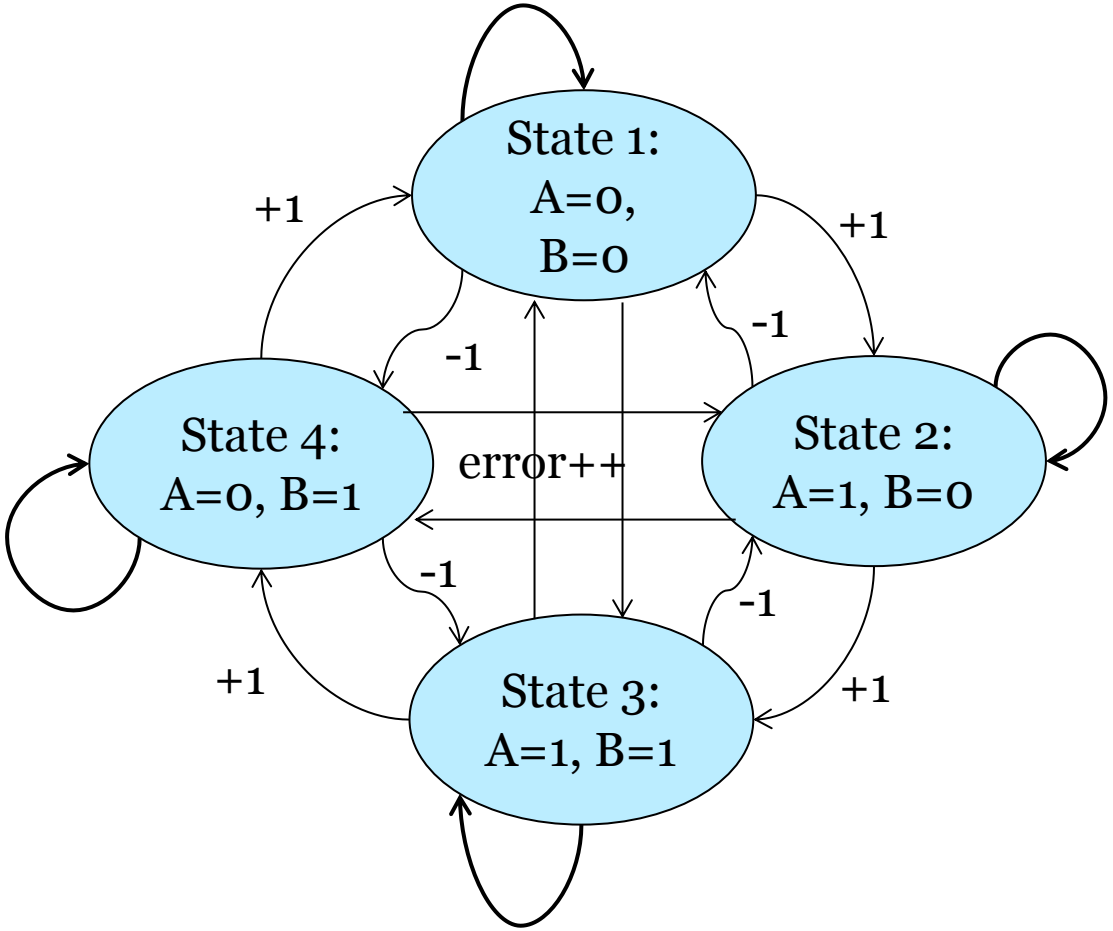
- Single counter no good for detecting direction of motion
- A pair of light sources/detectors phased ¼ cycle apart ("in quadrature") will detect direction, need up/down counter



FSM – Practical Example

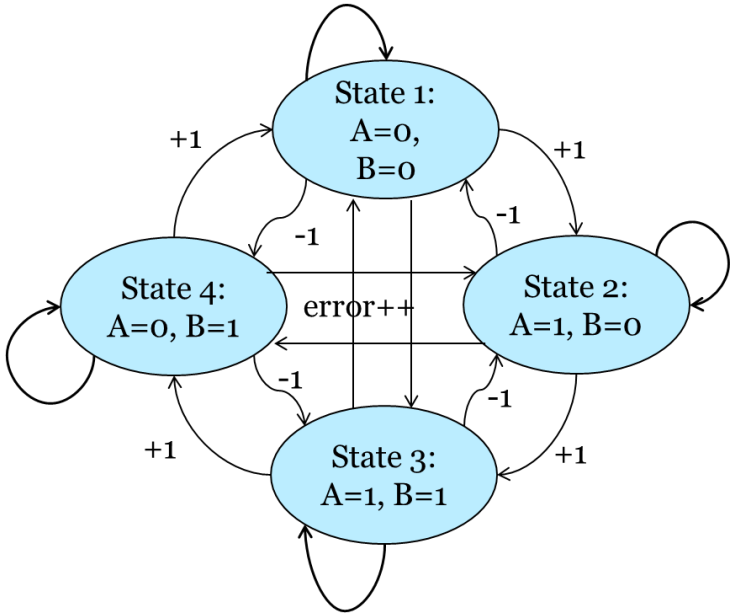
Construct a state transition diagram





FSM – Practical Example

- Can now construct a state machine according to the usual architecture
- Your job will be to:
  - Define **four states**
  - Fill in the logic of the state machine structure so that the code correctly:
    - **counts up or down** as it receives the pulses on channels A and B, and
    - **detects error** if pulses out of sequence
    - measure movement which occurred!



switch vs if / else if / else

if, else if .. else	switch

```
if ( c == 'a' )
{
   printf("Hi");
}
else if ( c == 'b' )
{
   printf("Bye");
}
else
{
   printf("Err")
}
```

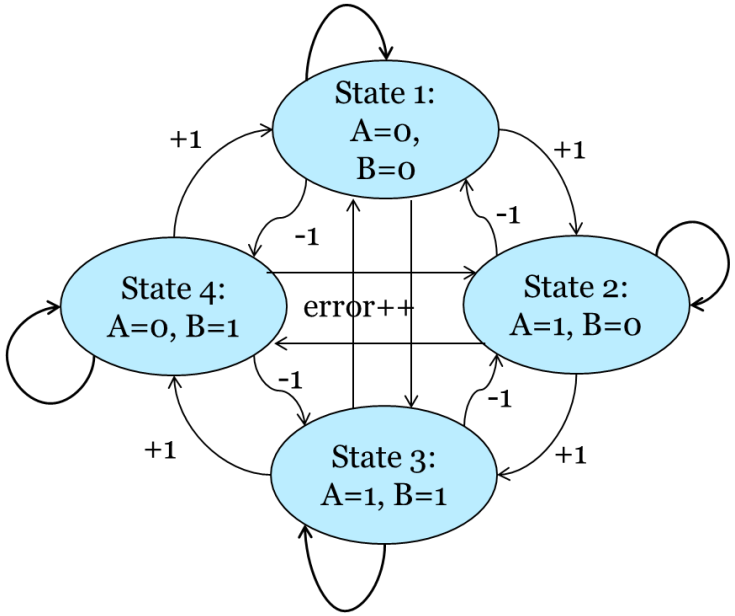
```
switch (c)
{
    case 'a' :  printf("Hi");
                break;
    case 'b' :  printf("Bye");
                break;
    default :   printf("Err");
                break;
}
```

Note: {} are optional here as only one line of code of controlled by each condition

FSM – Practical Example

```
channelAState = digitalRead(channelA);
channelBState = digitalRead(channelB);

switch (state)
{
    case state1:
    if (channelAState && !channelBState)
    {
        count+
    {
        count--
        state = state4;
    }
```

Your turn to do the remaining code!



switch vs if / else if / else

if, else if .. else	switch

`if ( c == 'a' )`
`{`
`  printf("Hi");`
`}`
`else if ( c == 'b' )`
`{`
`  printf("Bye");`
`}`
`else`
`{`
`  printf("Err")`
`}`
	`switch (c)`
	`{`
	`  case 'a' : printf("Hi");`
	`  break;`
	`  case 'b' : printf("Bye");`
	`  break;`
	`  default : printf("Err");`
	`  break;`
	`}`

Note: {} are optional here as only one line of code of controlled by each condition

Lab 1: Signals and Sensors
Link to Lectures

Laboratory 1: Signals and Sensors – Link to Lectures

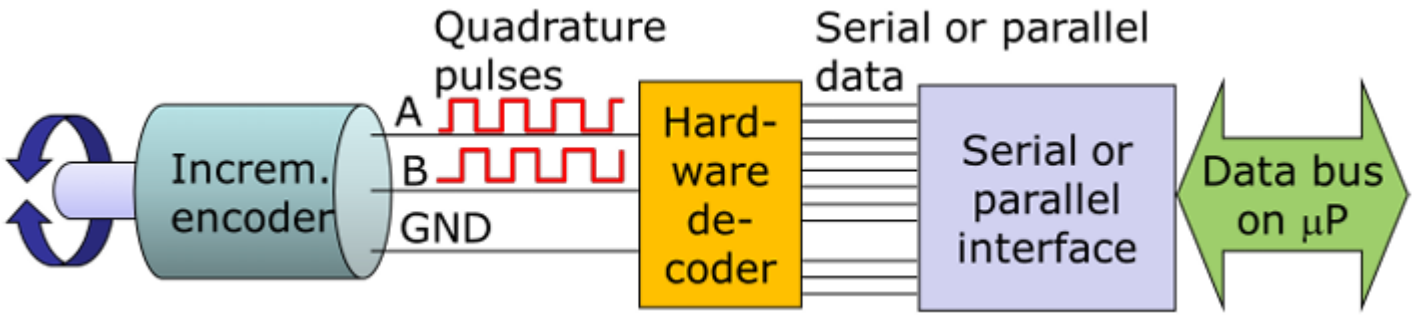
As explained in the introductory exercise, the overall objectives of this laboratory are:

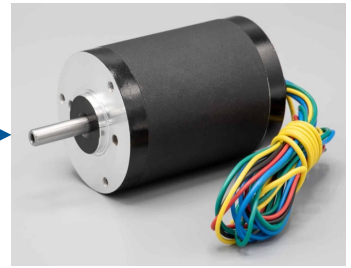
- To provide practical experience of *interfacing* a variety of *signal sources and sensors* to the Arduino Mega 2560 using various peripheral boards
- To give further experience of using the Arduino's dialect of the *C language*
- To provide experience of the interfacing of a servo motor and *incremental encoder* using an *H bridge*, the *LS7366R up/down counter* and the Arduino
- To enable students to *see in practice* the waveforms associated with *pulse width modulation* and quadrature encoding.
- To give a "sneak preview" of the use of *finite state machines* and of the use of *interrupts*

Laboratory 1: Signals and Sensors – Link to Lectures

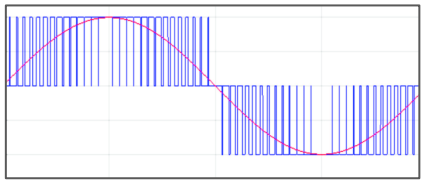
Encoder

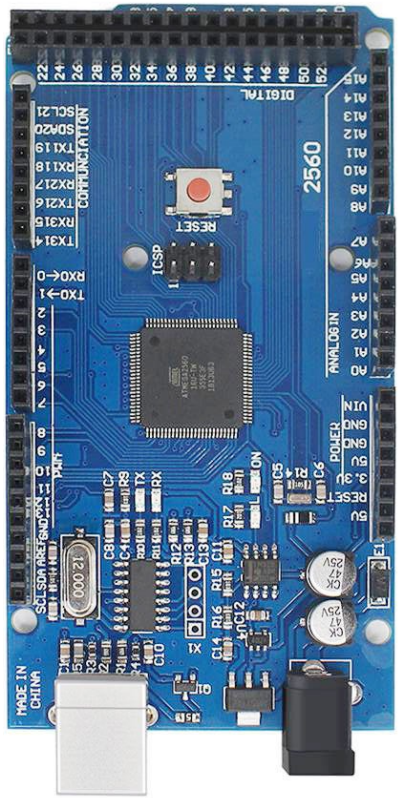
Digital Signal





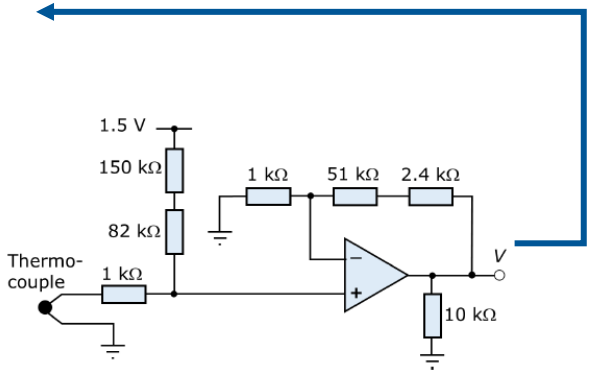
Motor Driver





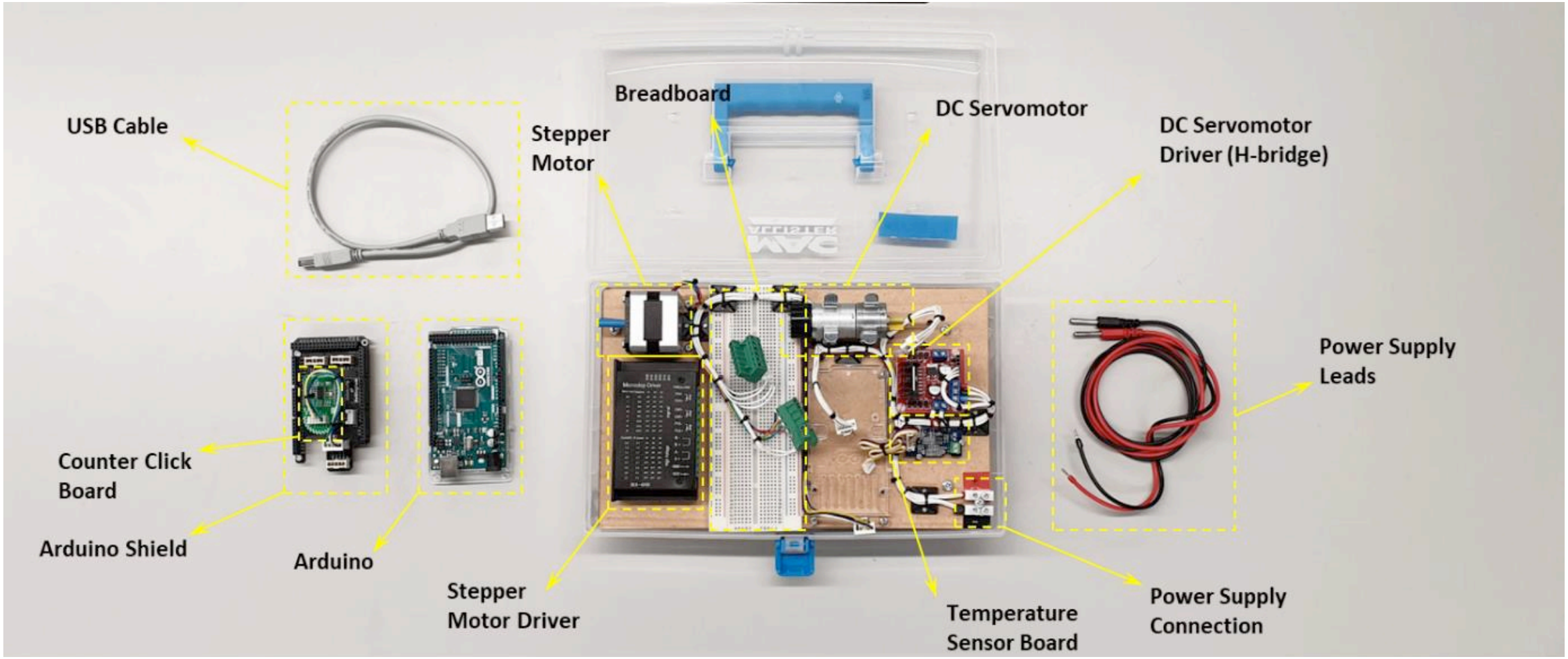
Thermocouple

Analog Signal



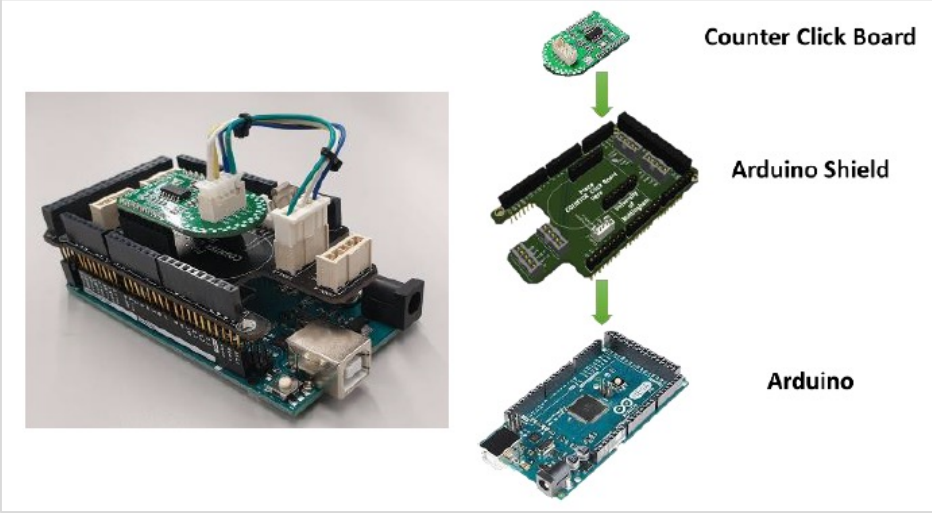
Laboratory 1: Signals and Sensors – Link to Lectures

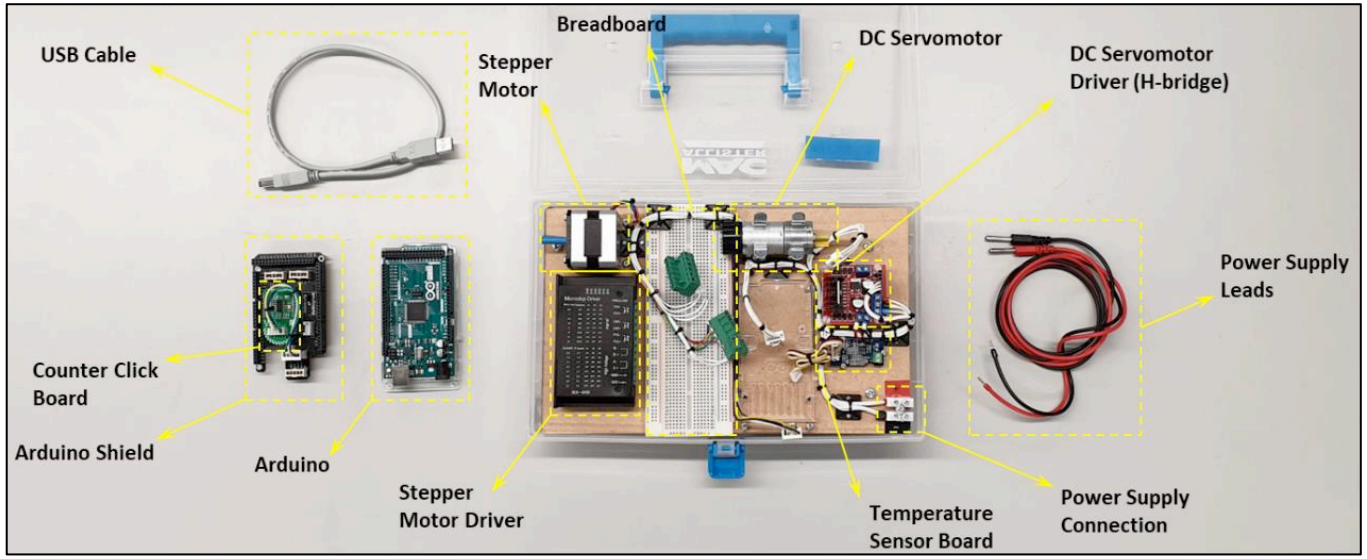
- This is the lab kit which you will use in Lab 1 (different from the "Take-Home" kits)
- More details will be given in the lab!
- Here, I like to focus on few components!

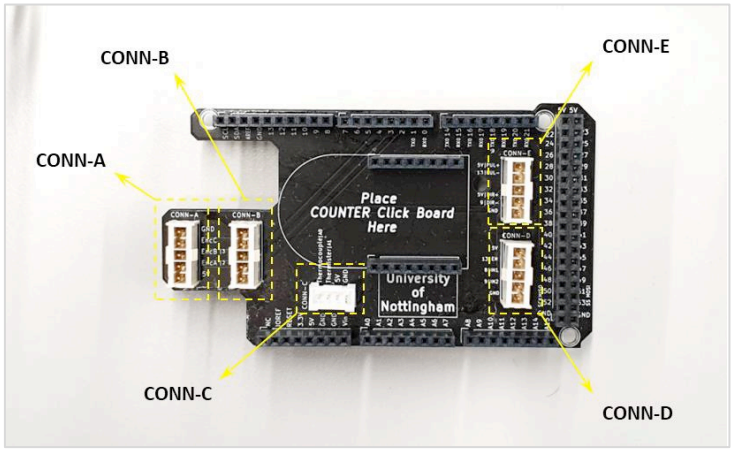


Laboratory 1: Signals and Sensors – Link to Lectures

- This is the lab kit which you will use in Lab 1 (different from the "Take-Home" kits)
- More details will be given in the lab!
- Here, I like to focus on few components!



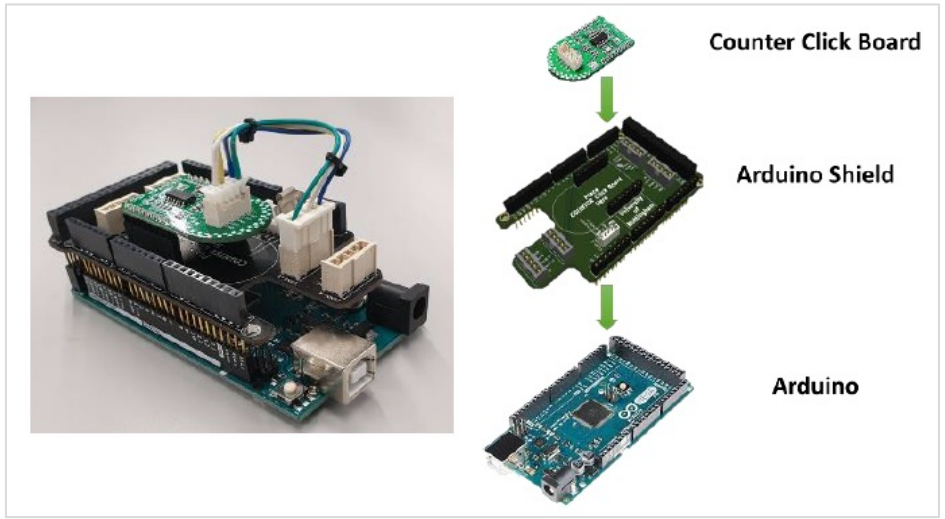


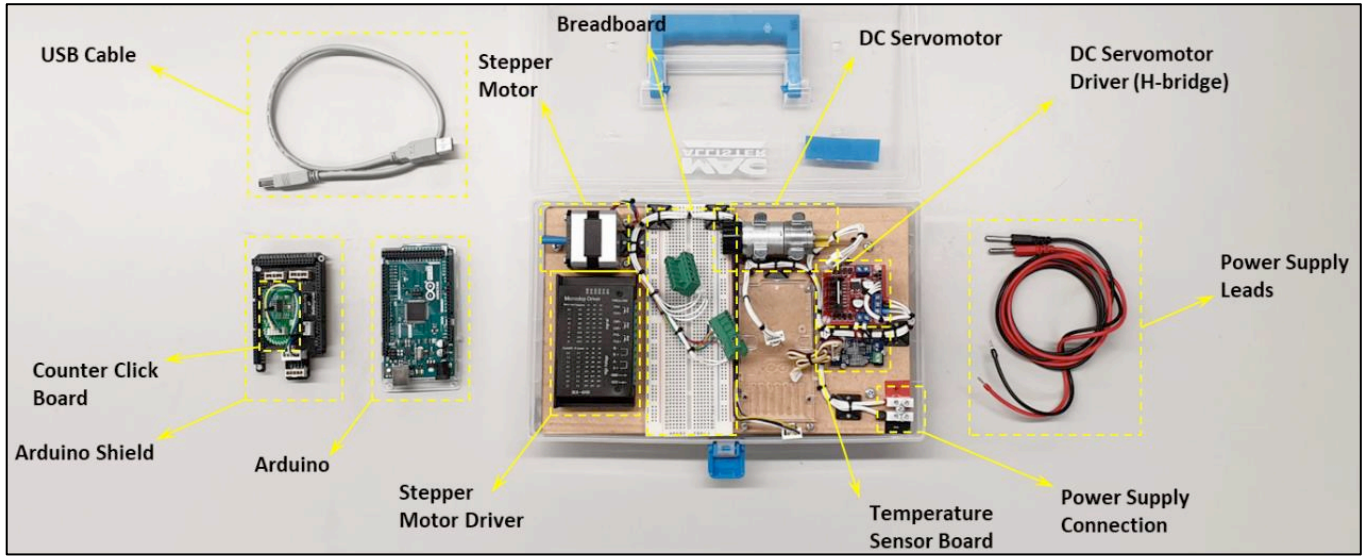


Arduino Shield – Hardware interface that has been specifically developed to work for these experiments. All connections you require to make with the Arduino (for these experiments) can be made via connectors (no bread board and jumper wires!). The Shield slots on top of the Arduino.

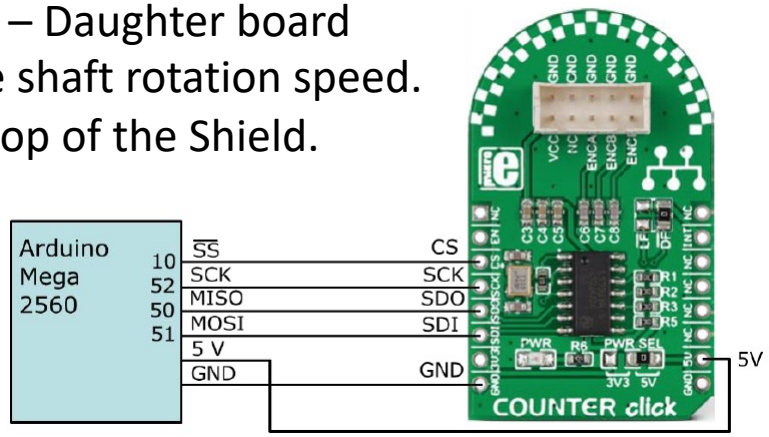
Laboratory 1: Signals and Sensors – Link to Lectures

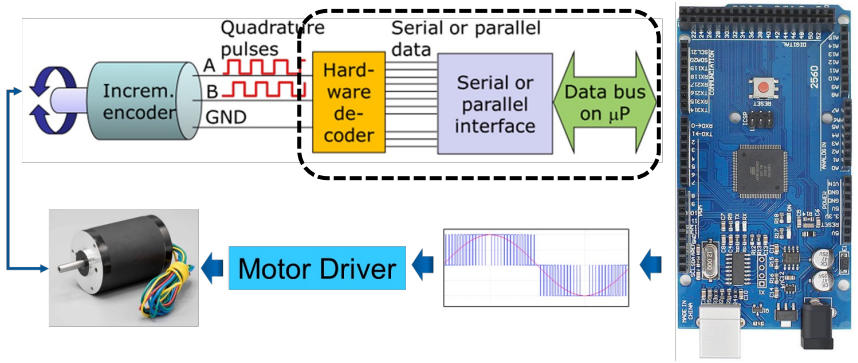
- This is the lab kit which you will use in Lab 1 (different from the "Take-Home" kits)
- More details will be given in the lab!
- Here, I like to focus on few components!





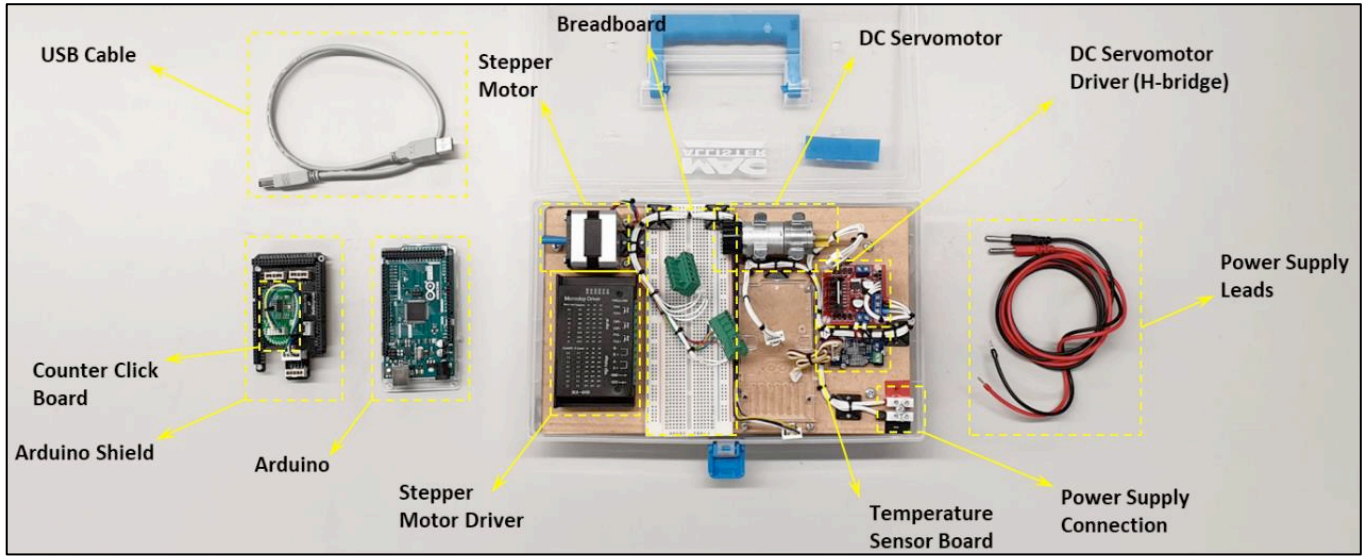
Counter Click Board – Daughter board required to measure shaft rotation speed. This board slots on top of the Shield.





Laboratory 1: Signals and Sensors – Link to Lectures

- This is the lab kit which you will use in Lab 1 (different from the "Take-Home" kits)
- More details will be given in the lab!
- Here, I like to focus on few components!

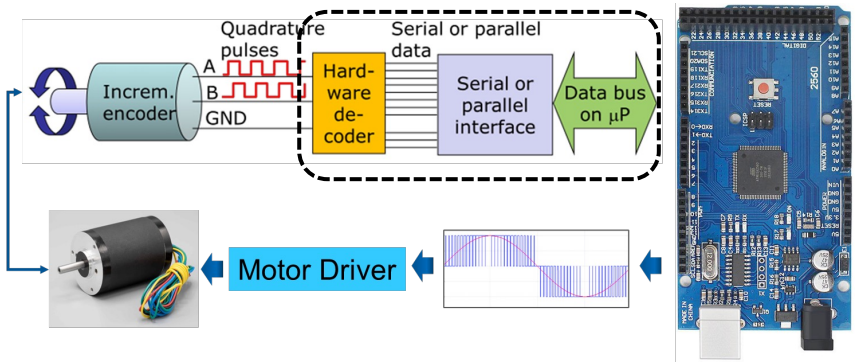


So far, we learned two methods to count pulses

- Polling
  - Arduino language *(Lecture 1/2)*
  - Registers *(Lecture 2)*
- Timer/Counter
  - T/Cs on the Arduino *(Lecture 3)*
  - External T/Cs boards *(Lecture 3)*

In this lab, we will use three of these methods:

a. Using the Counter Click Board via the serial communication.
b. Using Arduino Timer to count Encoder Channel A pulses
c. Using your State Machine code



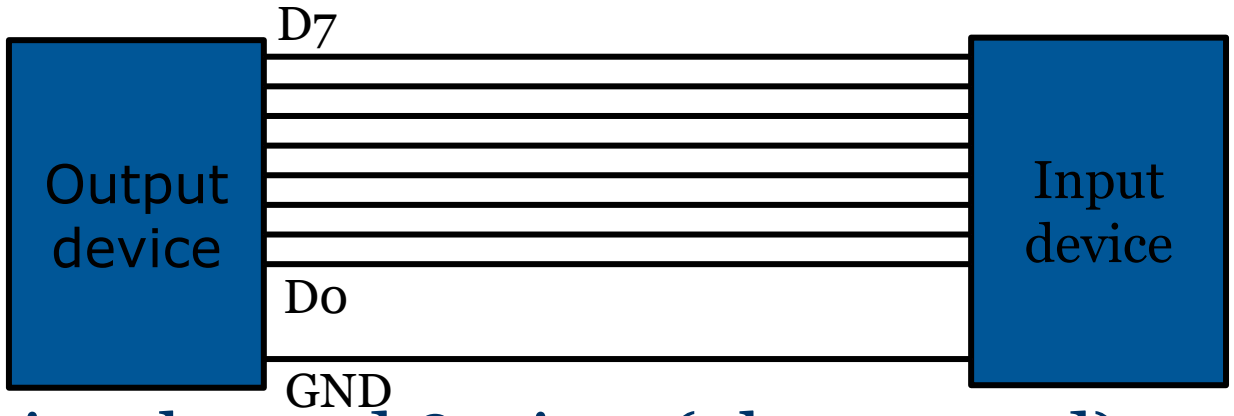
Which is which?! ☺

University of Nottingham
UK | CHINA | MALAYSIA

Serial Data Transfer

Serial Data vs. Parallel Data

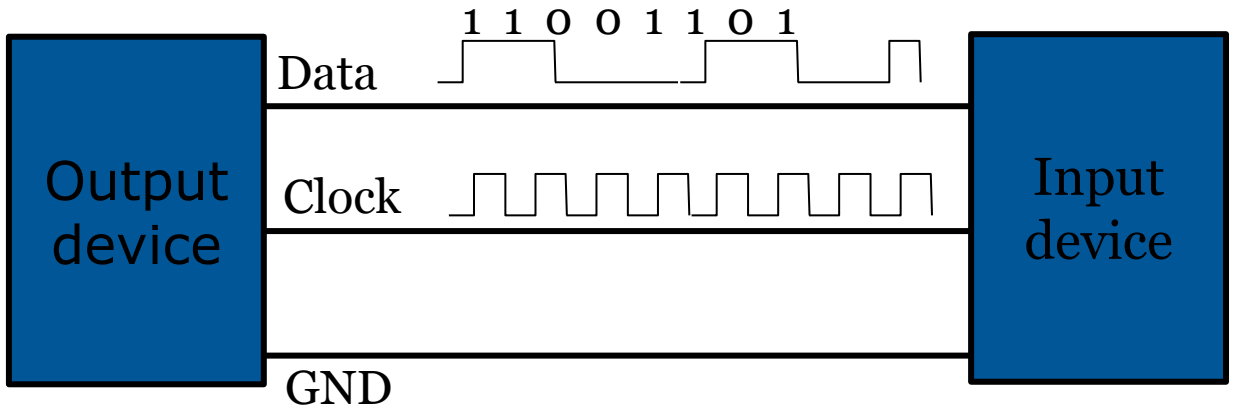
- So far know how to transfer single bits or (for example) an 8-bit number all bits at a time: **parallel data**



- Obviously need 8 wires (plus ground)
- OK for short distances, impractical for long (lots of wires, interference etc.)

Serial Data vs. Parallel Data

- In many situations it is more practical to send the data *one bit at a time* down only a *single wire plus ground*
- Some approaches need more wires for synchronisation and/or "handshaking"

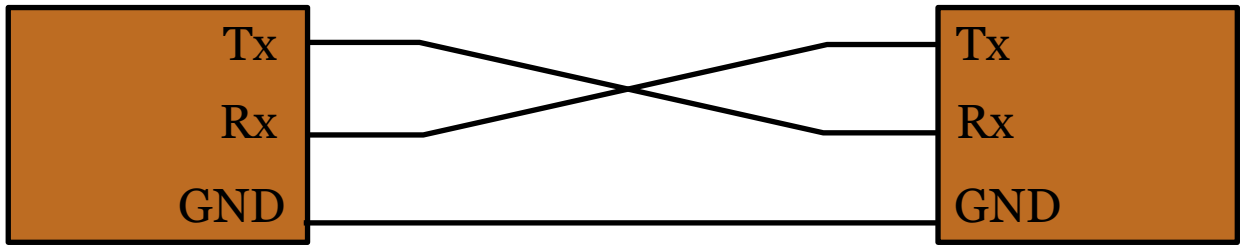


Serial data protocols

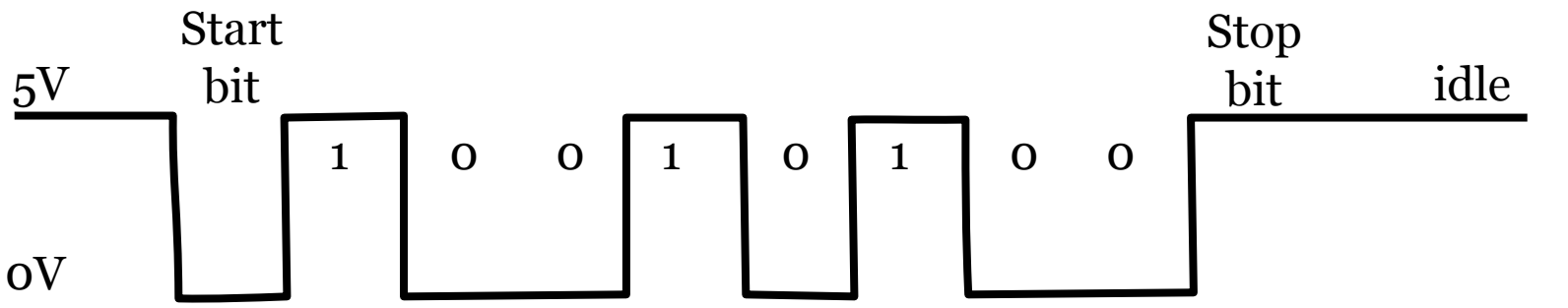
- Many serial data transfer protocols:
  - USB (various versions e.g., USB3.2)
  - RS232 (runs at typically ±12 V, historical standard for terminals etc.). 2 wires (Tx, Rx), asynchronous, bit rate ("baud rate") must be specified
  - I2C: 2 wires plus GND, low cost, slow
  - SPI: uses 4 wires plus GND, quicker than I2C but needs more wiring
  - MIDI: baud rate 31250, optical isolated

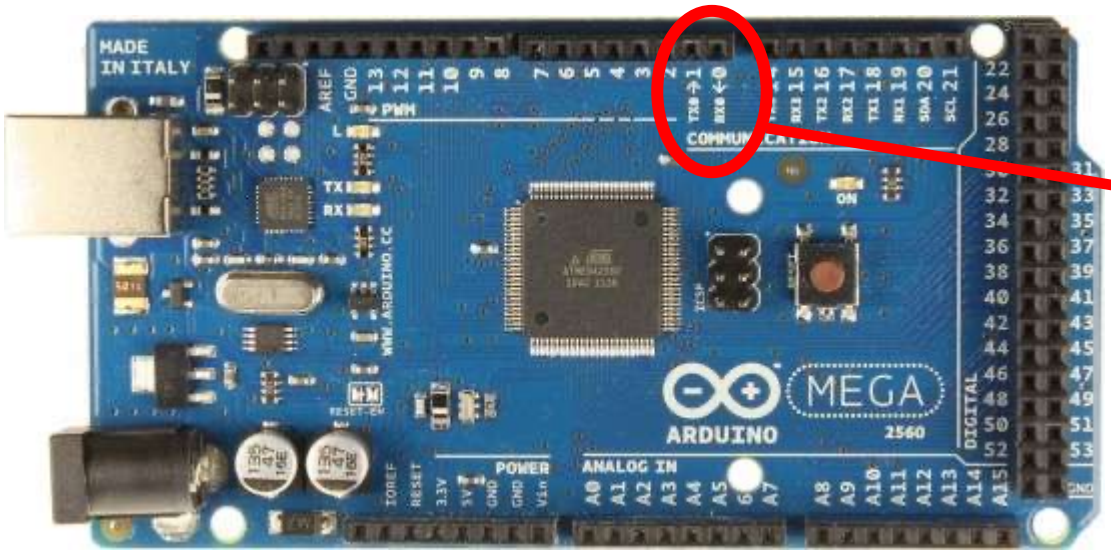
Asynchronous serial communication with the Arduino

- Achieved using *two pins* (typically 1 & 0): transmit (Tx) and receive (Rx) + GND
- Based on RS232 concept but uses 5V (~TTL) voltage levels
- No separate synchronisation signal: synchronised by synchronisation bit at start, agreed baud rate (here = bit rate)



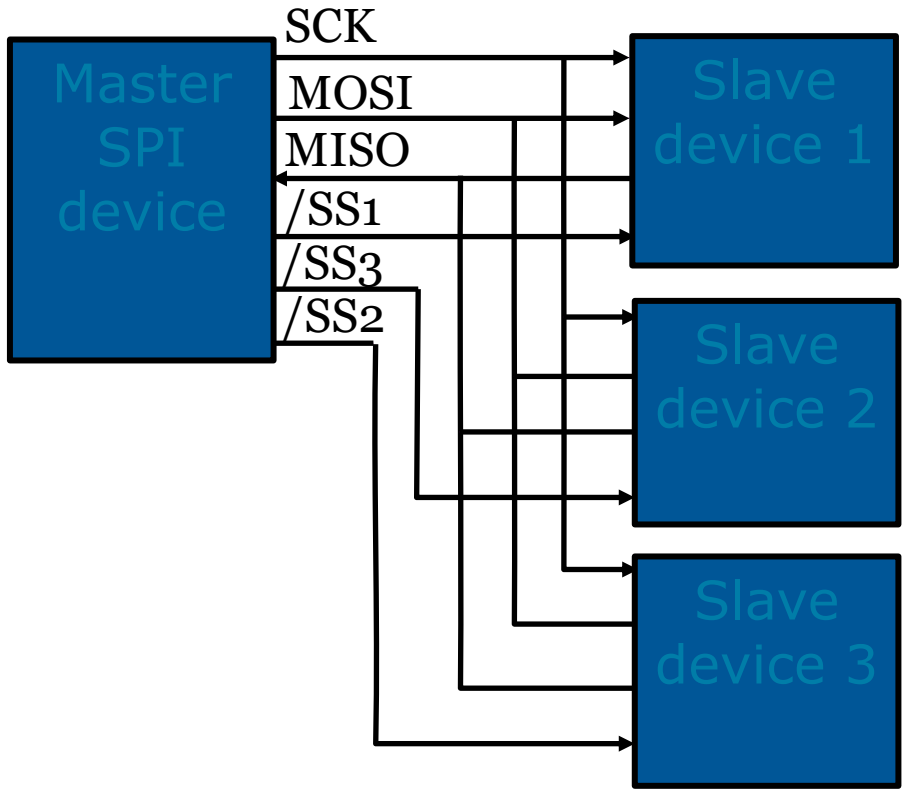
Asynchronous serial communication with the Arduino





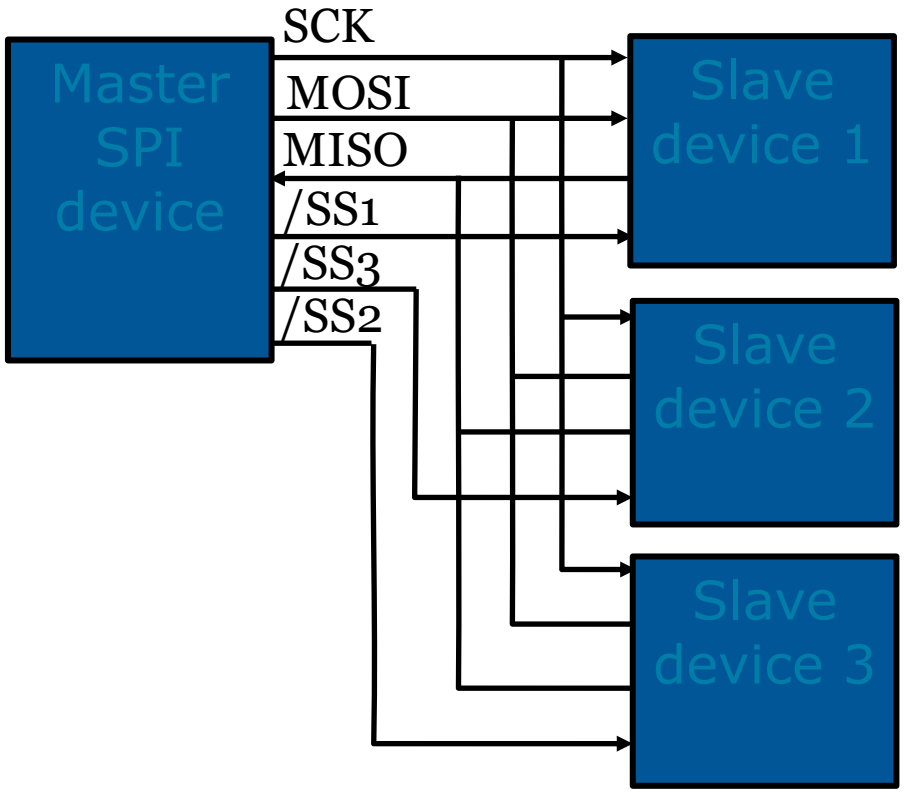
Serial Peripheral Interface (SPI): example of synchronous serial

- One “master” device can communicate both ways with multiple “slave” devices e.g.
  - Another Arduino
  - A specialist counter e.g., for quadrature signals (Lab 1!)
  - An LCD display



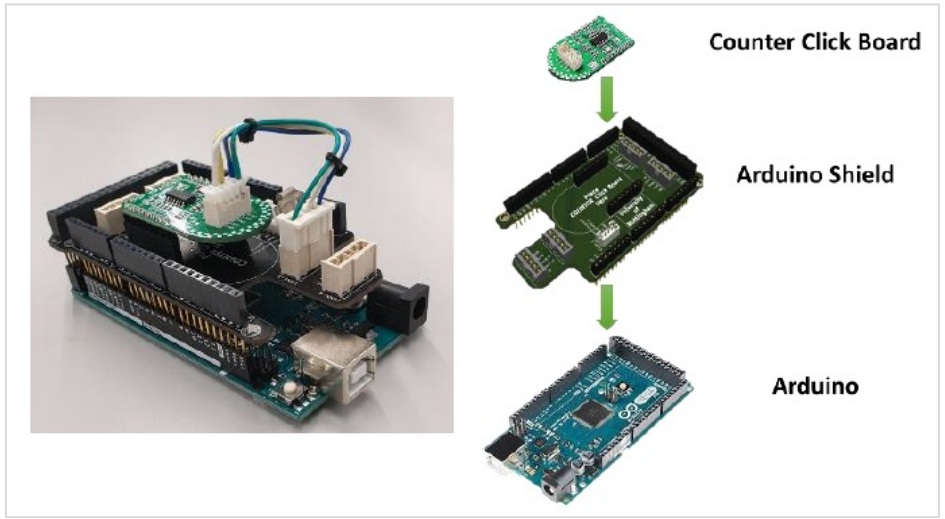
Serial Peripheral Interface (SPI): example of synchronous serial

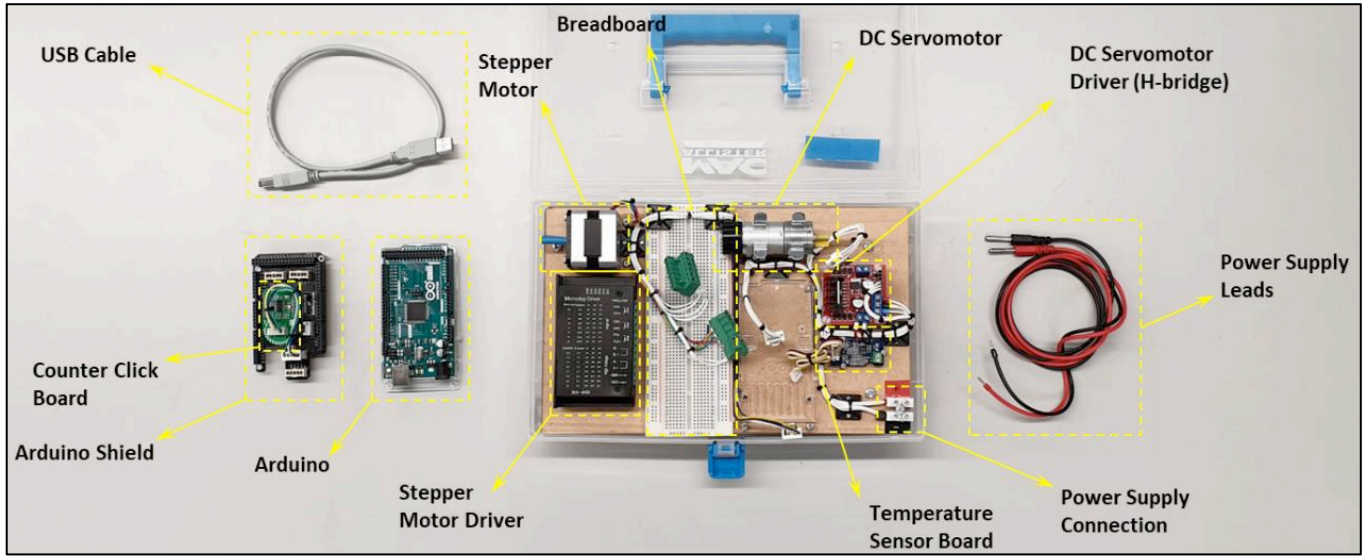
- What are the wires?
- *Three are common* between all slaves:
  - Master input slave output (MISO)
  - Master output slave input (MOSI)
  - Clock (SCK, timing pulses)
- Each slave has slave select (SS) line: goes LOW to *select* slave



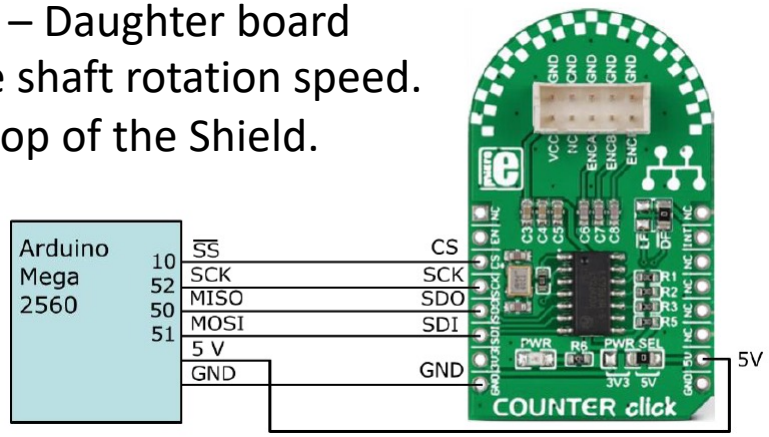
Laboratory 1: Signals and Sensors – Link to Lectures

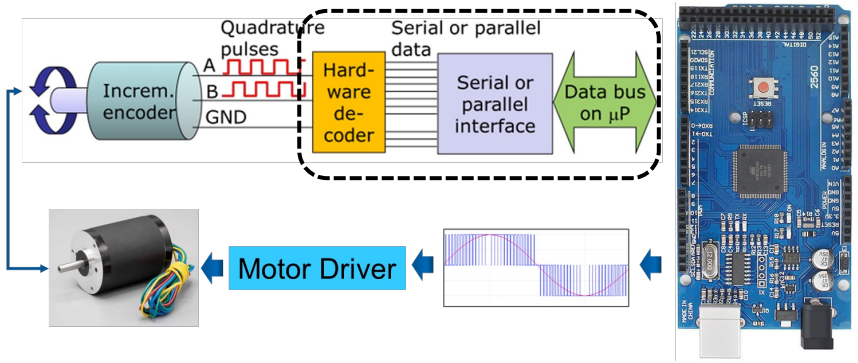
- This is the lab kit which you will use in Lab 1 (different from the "Take-Home" kits)
- More details will be given in the lab!
- Here, I like to focus on few components!





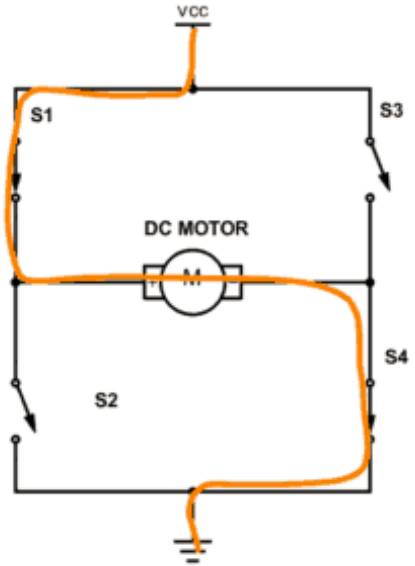
Counter Click Board – Daughter board required to measure shaft rotation speed. This board slots on top of the Shield.

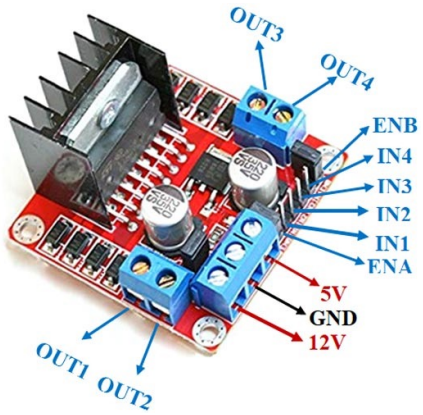


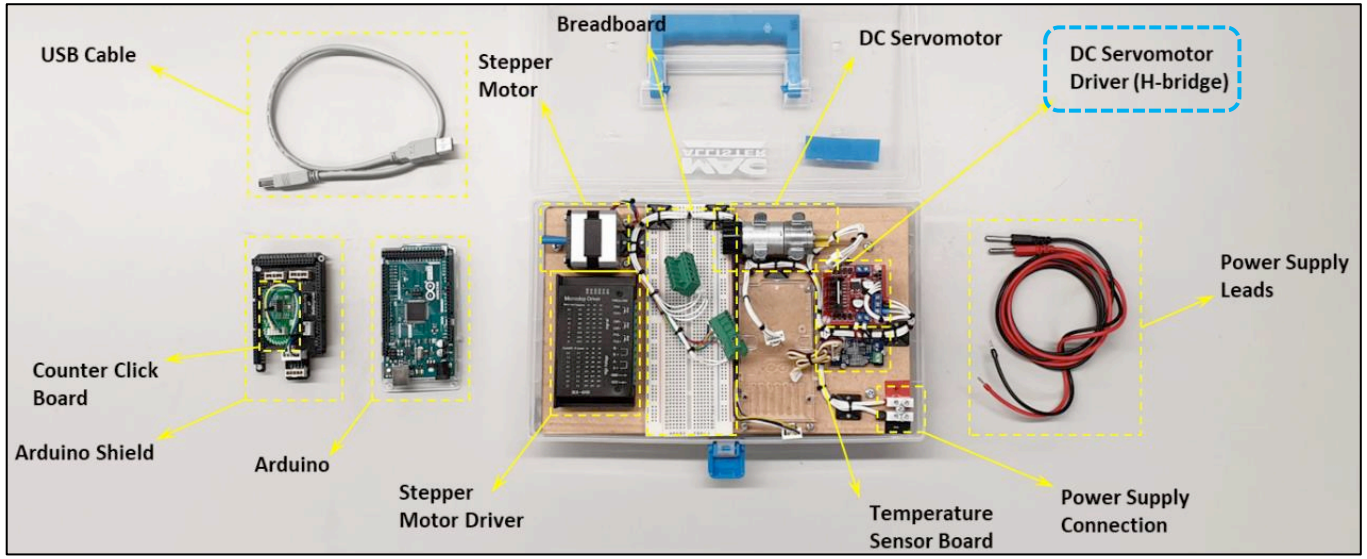


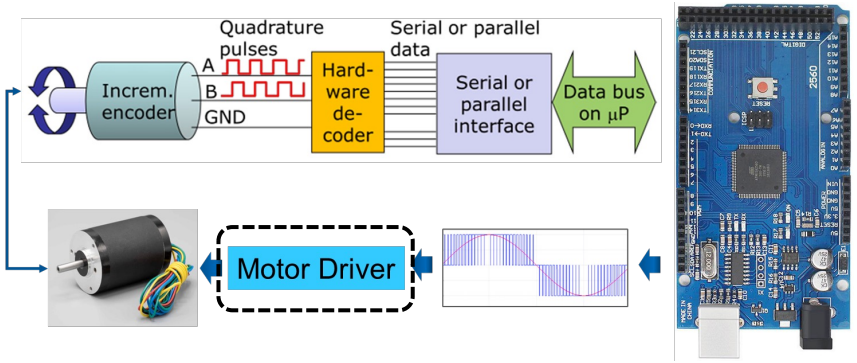
Laboratory 1: Signals and Sensors – Link to Lectures

- This is the lab kit which you will use in Lab 1 (different from the "Take-Home" kits)
- More details will be given in the lab!
- Here, I like to focus on few components!





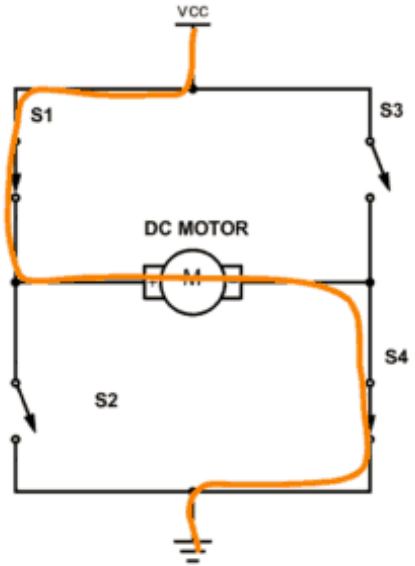


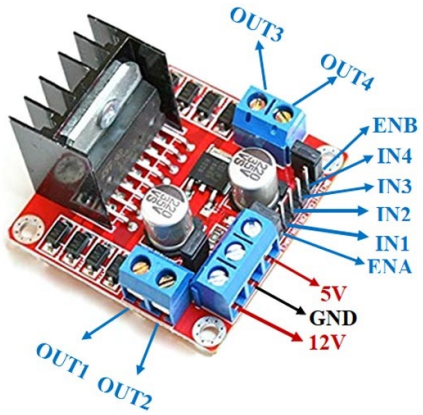


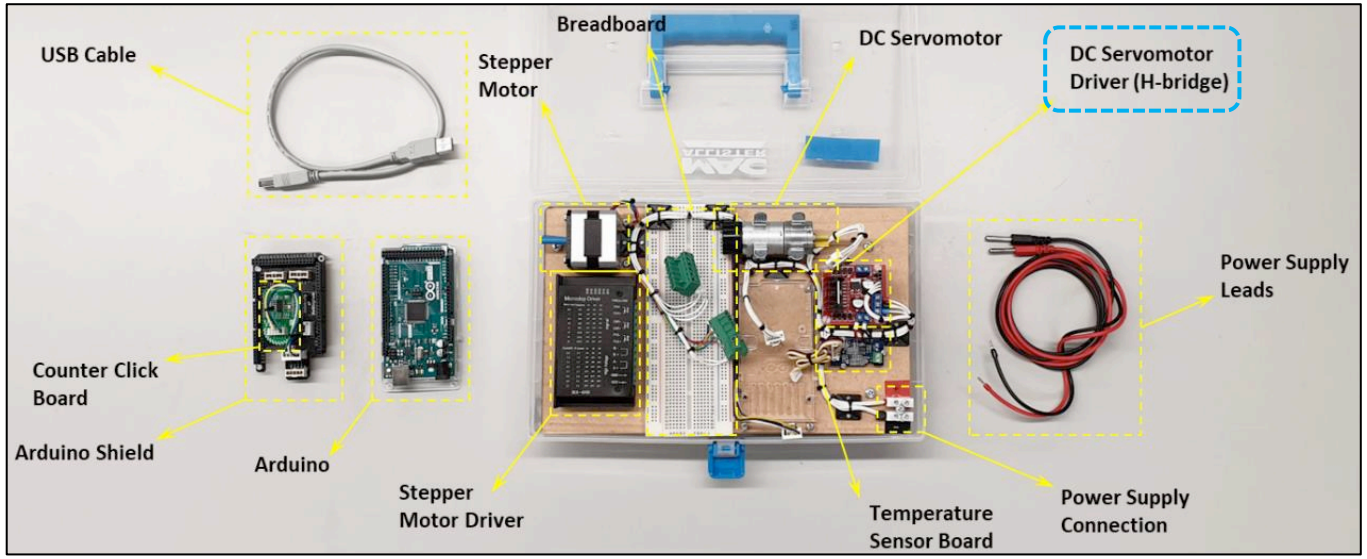
DC Servomotor Driver (H-Bridge) – Controls the DC Servomotor. A constant DC voltage across the motor's terminals would also spin the motor, but you cannot control direction or torque produced. The H-Bridge allows that.

Laboratory 1: Signals and Sensors – Link to Lectures

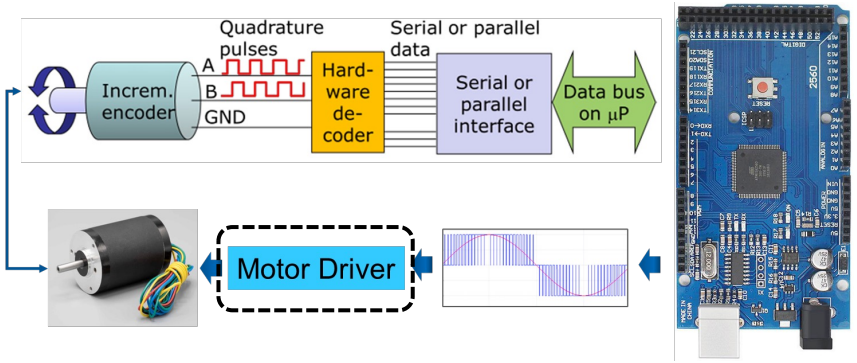
- This is the lab kit which you will use in Lab 1 (different from the "Take-Home" kits)
- More details will be given in the lab!
- Here, I like to focus on few components!







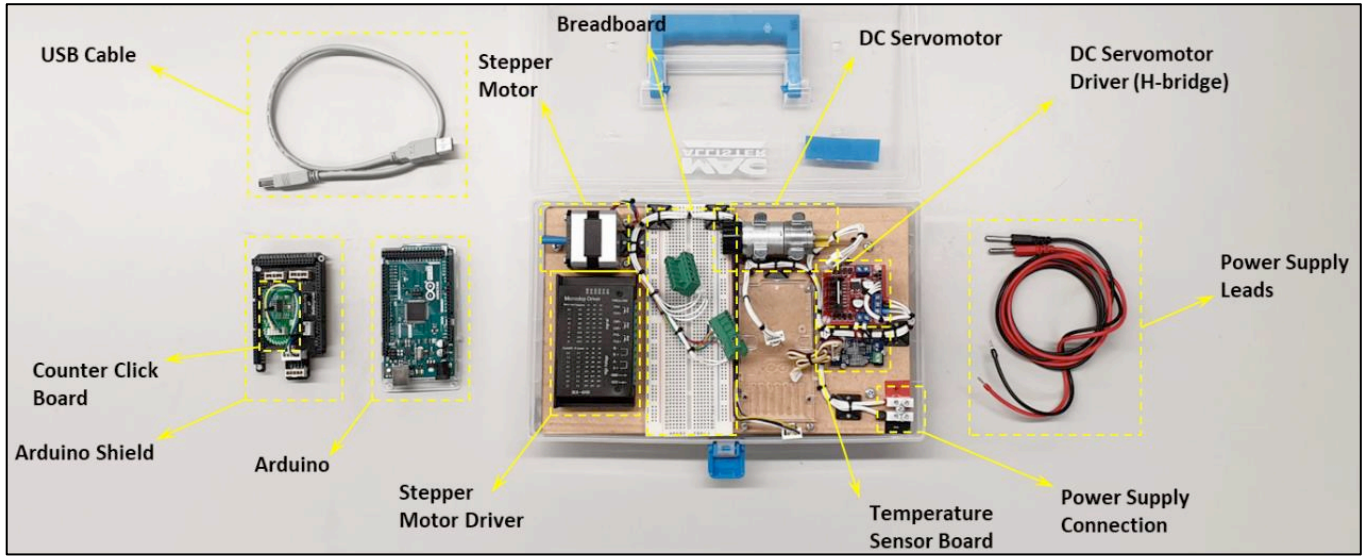
We can generate PWM using T/Cs!
Hard? ☺ No worries!
We will use an easy way



analogWrite(Pin, val); // val from 0 to 255

Laboratory 1: Signals and Sensors – Link to Lectures

- This is the lab kit which you will use in Lab 1 (different from the "Take-Home" kits)
- More details will be given in the lab!
- Here, I like to focus on few components!

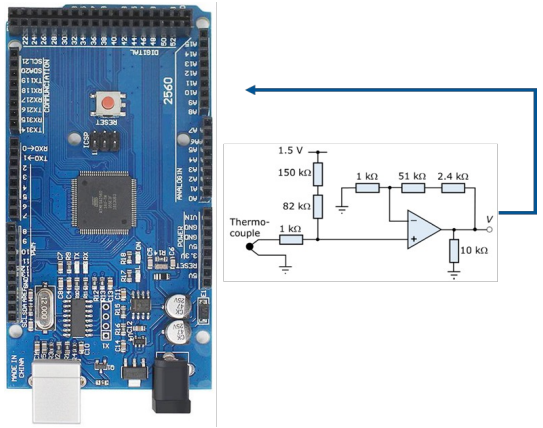


Also, for the thermocouple, we will use the "sister" function of

analogWrite(Pin, val); // val from 0 to 255

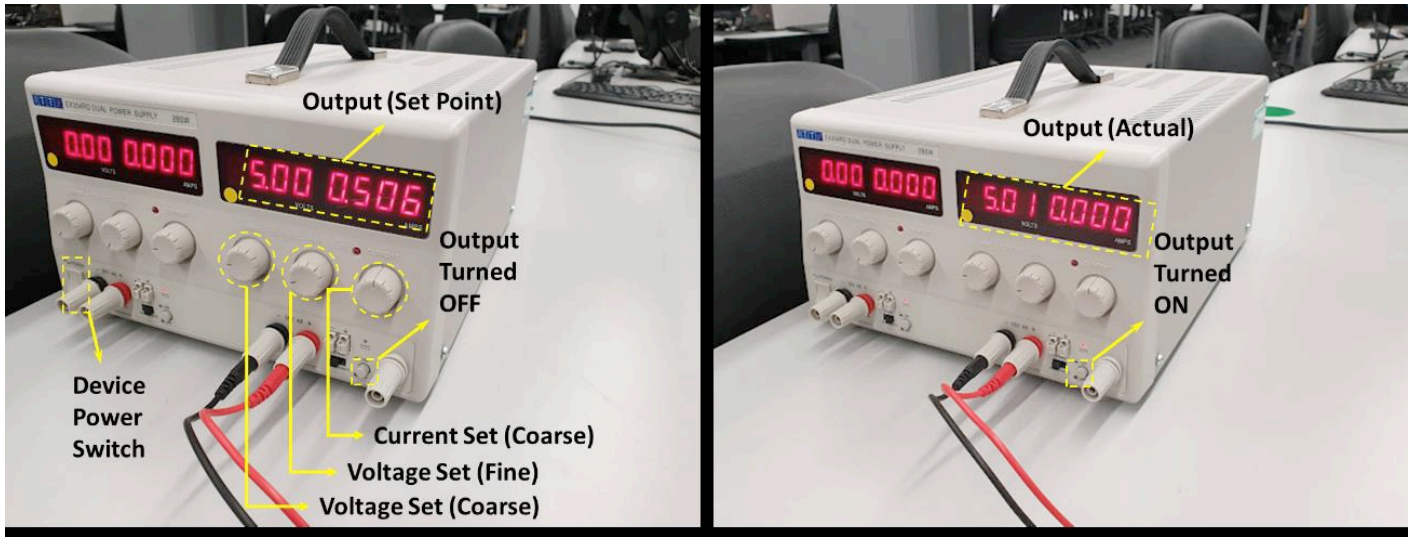
Which is

val =analogRead(Pin); // val from 0 (0v) to 1023 (5v)

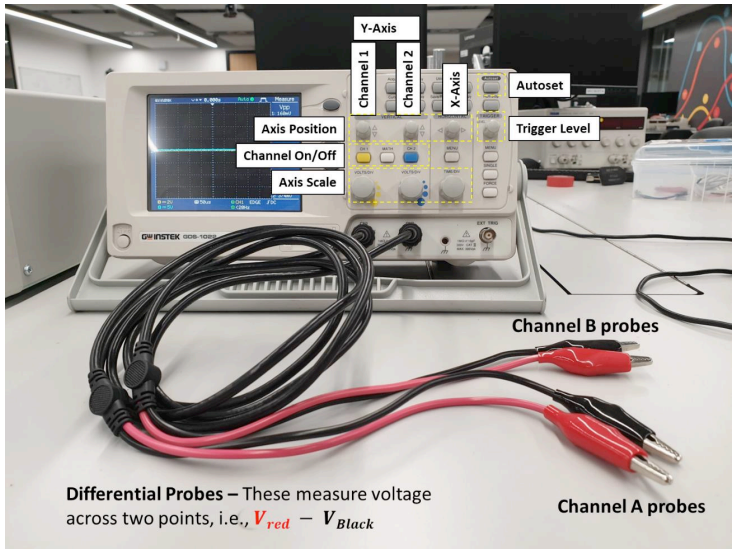


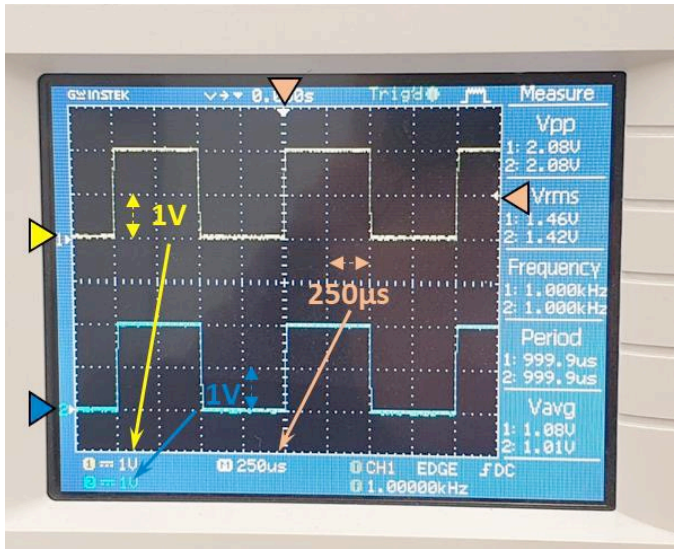
Laboratory 1: Signals and Sensors – Link to Lectures

Power Supply



Oscilloscope






University of
Nottingham
UK | CHINA | MALAYSIA
Have a Look Into the Lab Code!

Setup and Initialisation

```
/* Configure Timer 5 to count pulses on pin 47 */
pinMode(47, INPUT_PULLUP);           // set pin to input with pullup resistor

TCCR5A = 0; // No waveform generation needed.
TCCR5B = (1<<CS50) | (1<<CS51) | (1<<CS52); // Normal mode, clock from pin T5 on rising edge. T5 is Arduinos Pin 47
TCCR5C = 0; // No force output compare.
TCNT5 = 0;  // Initialise counter register to zero.
TIMSK5= (1<<TOIE5);  // Enable overflow interrupt
sei();      // Enable all interrupts
bigLaps = 0; // Initialise number of overflows
```

```
/* Encoder input pins (used for state machine and interrupts) */
#define channelA 2
#define channelB 3
```

```
/* Set encoder pins as input but with pullup resistors*/
pinMode(channelA, INPUT_PULLUP);
pinMode(channelB, INPUT_PULLUP);
```

```
/* Pins used for L298 driver */
#define enA 13      /* PWM output, also visible as LED */
#define in1 8       /* H bridge selection input 1 */
#define in2 9       /* H bridge selection input 2 */
#define minPercent -100.0
#define maxPercent 100.0
```

```
/* Configure control pins for L298 H bridge */
pinMode(enA, OUTPUT);
pinMode(in1, OUTPUT);
pinMode(in2, OUTPUT);
```

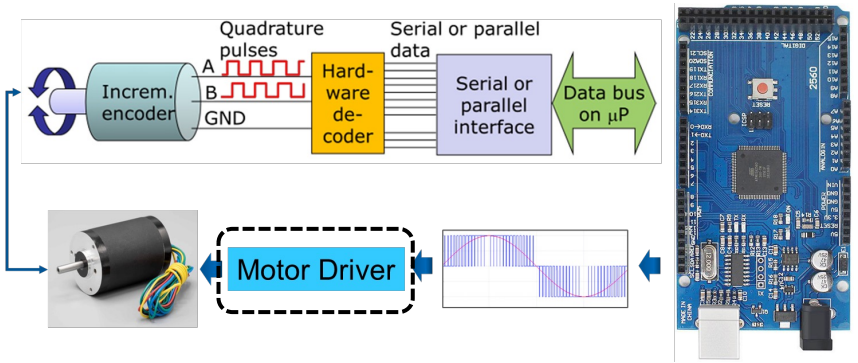
```
/* Used for state machine and encoder reading */
typedef enum states{state1=1, state2, state3, state4};
volatile long int count = 0;
volatile long int error = 0;
volatile states state;
bool channelAState, channelBState;
```

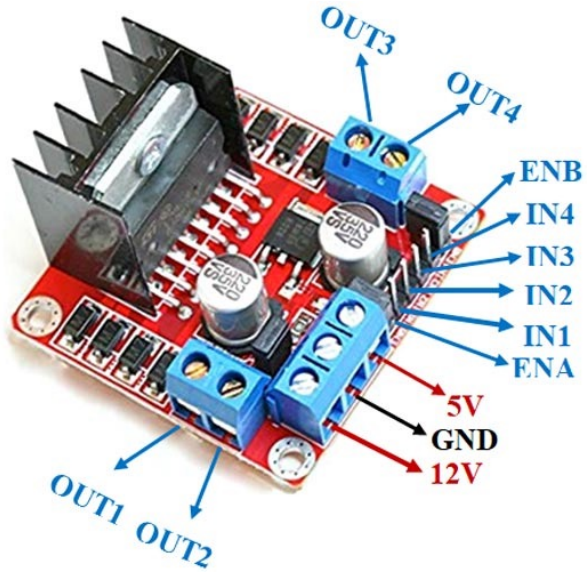
```
//attachInterrupt(digitalPinToInterrupt(channelA), updateEncoderStateMachine, CHANGE);
//attachInterrupt(digitalPinToInterrupt(channelB), updateEncoderStateMachine, CHANGE);
```

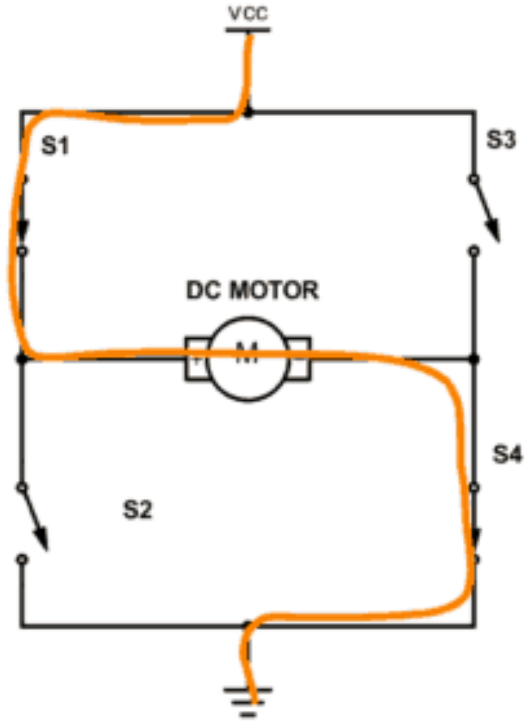
This is commented, we will come to it later!

Inside the Loop() Function!

```
void driveMotorPercent(double percentSpeed)
/* Output PWM and H bridge signals based on positive or negative duty cycle %
*/
{
      percentSpeed = constrain(percentSpeed, -100, 100);
      int regVal = map(percentSpeed, -100, 100, -255, 255);
      analogWrite(enA, (int)abs(regVal));
      digitalWrite(in1, regVal>0);
      digitalWrite(in2, !(regVal>0));
}
```





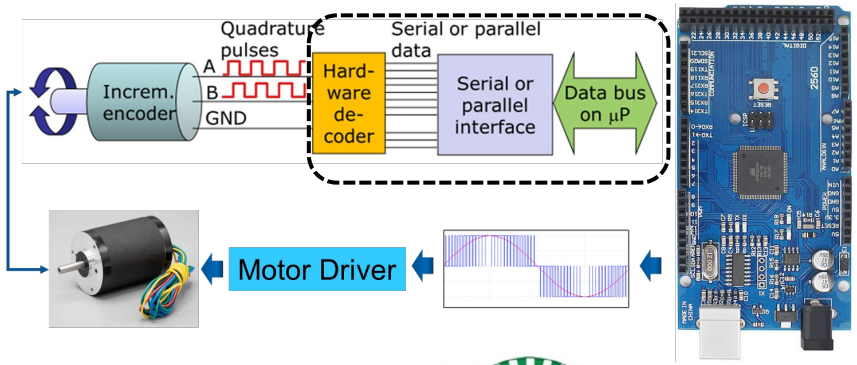


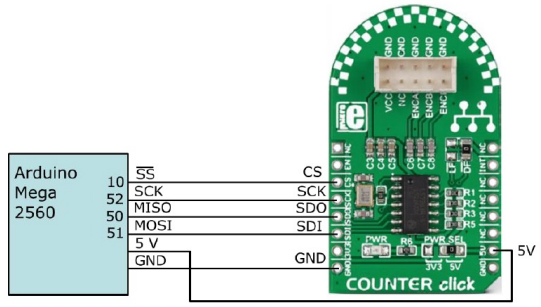
University of Nottingham
UK | CHINA | MALAYSIA

Inside the Loop() Function!

```
void SetUpLS7366RCounter(void)
/* Initialiseds LS7366R hardware counter on Counter Click board to read quadrature
signals */
{
    /* Control registers in LS7366R - see LS7366R datasheet for this and subsequent
control words */
    unsigned char IR = 0x00, MRD0=0x00;

    // SPI initialization
    SPI.begin();
    //SPI.setClockDivider(SPI_CLOCK_DIV16);      // SPI at 1Mhz (on 16Mhz clock)
    delay(10);

   /* Configure as free-running 4x quadrature counter */
   digitalWrite(chipSelectPin,LOW); /* Select chip and initialise transfer */
   /* Instruction register IR */
   IR |= 0x80;   /* Write to register (B7=1, B6=0) */
   IR |= 0x08;   /* Select register MDR0: B5=0, B4=0, B3=1 */
   SPI.transfer(IR); /* Write to instruction register */
   /* Mode register 0 */
   MRD0 |= 0x03;    /* 4x quadrature count: B0=1, B1=1 */
   /* B2=B3=0: free running.  B4=B5=0: disable index. */
   /* B6=0: asynchronous index.  B7: Filter division factor = 1. */
   SPI.transfer(MRD0);
   digitalWrite(chipSelectPin,HIGH);

   /* Clear the counter i.e. set it to zero */
   IR = 0x00; /* Clear the instructino register IR */
   digitalWrite(chipSelectPin,LOW); /* Select chip and initialise transfer */
   IR |= 0x20; /* Select CNTR: B5=1,B4=0,B3=0; CLR register: B7=0,B6=0 */
   SPI.transfer(IR); /* Write to instruction register */
   digitalWrite(chipSelectPin,HIGH);

}
```

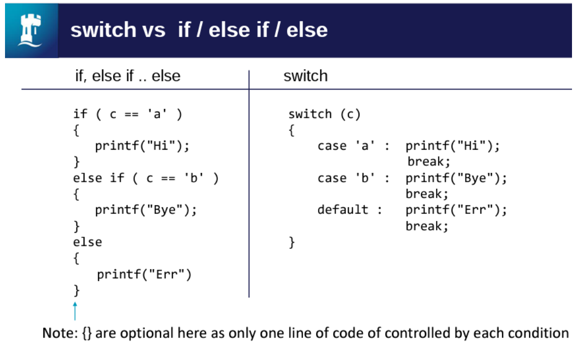
```
long readEncoderCountFromLS7366R()
/* Reads the LS7366R chip to obtain up/down count from encoder.  Reads four
   bytes separately then concverts them to a long integer using a union */
{
    fourBytesToLong converter; /* Union of four bytes and a long integer */
    digitalWrite(chipSelectPin,LOW); /* Make LS7366R active */
    SPI.transfer(0x60); // Request count
    converter.bytes[3] = SPI.transfer(0x00); /* Read highest order byte */
    converter.bytes[2] = SPI.transfer(0x00);
    converter.bytes[1] = SPI.transfer(0x00);
    converter.bytes[0] = SPI.transfer(0x00); /* Read lowest order byte */

    digitalWrite(chipSelectPin,HIGH); /* Make LS7366R inactive */
    return converter.result;
}
```





Inside the Loop() Function!

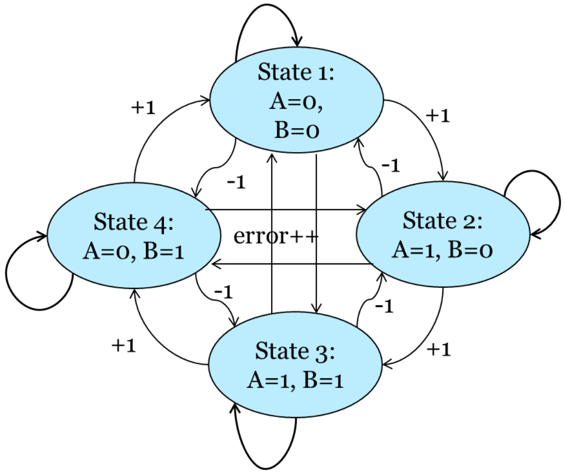
```
void updateEncoderStateMachine()
/* User written code to update state and increment count of state machine  */
{
  channelAState = digitalRead(channelA);
  channelBState = digitalRead(channelB);

  switch (state)
  {
     case state1:
     if (channelAState && !channelBState)
     {
        count++;
        state = state2;
     }
     /* else if .... a lot of code goes here! */
     /* don't forget "break" at end of each case. */
  }
}
```

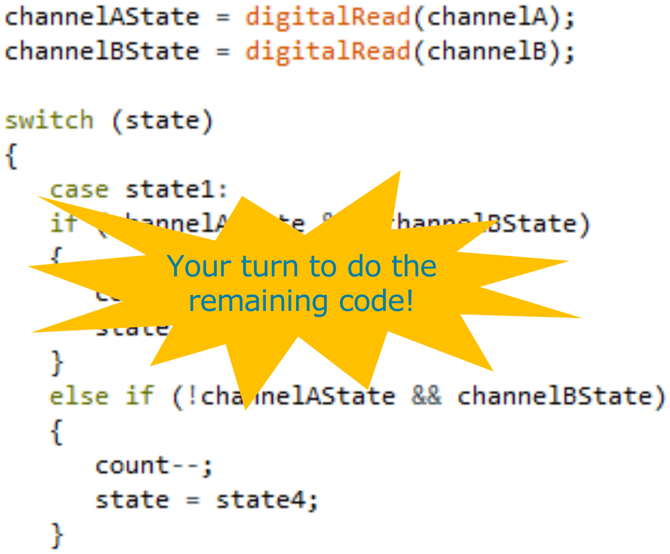


switch vs if / else if / else

if, else if .. else	switch
`if ( c == 'a' ) { printf("Hi"); } else if ( c == 'b' ) { printf("Bye"); } else { printf("Err") }`	`switch (c) { case 'a' : printf("Hi"); break; case 'b' : printf("Bye"); break; default : printf("Err"); break; }`

Note: {} are optional here as only one line of code of controlled by each condition





Inside the Loop() Function!

You have seen this before!

```
ISR(TIMER5_OVF_vect )
{
  //when this runs, you had 65536 pulses counted.
  bigLaps++;
}
```

Yes, it is the Interrupt Service Routine ISR

We enabled it like this!

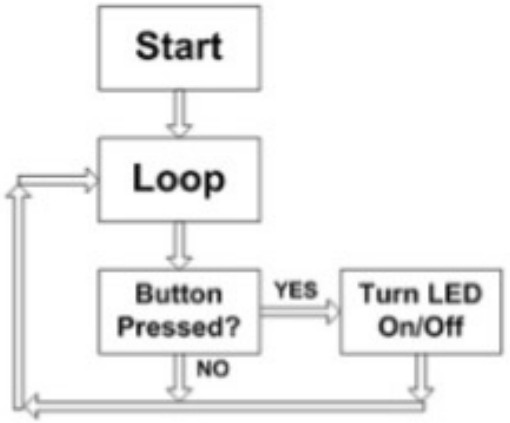
```
TIMSK5= (1<<TOIE5);  // Enable overflow interrupt
```

Basically, when T/C5 overflows, the µP will execute this routine/function, typically in a fraction of ms!

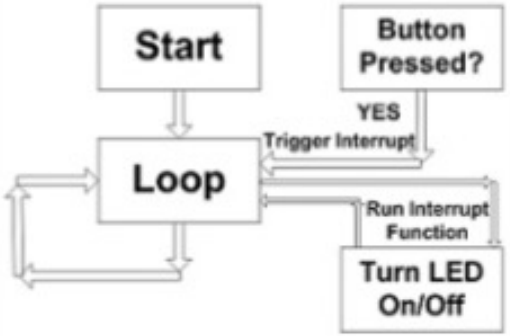

University of
Nottingham
UK | CHINA | MALAYSIA
What is an Interrupt?!

What is an Interrupt?!

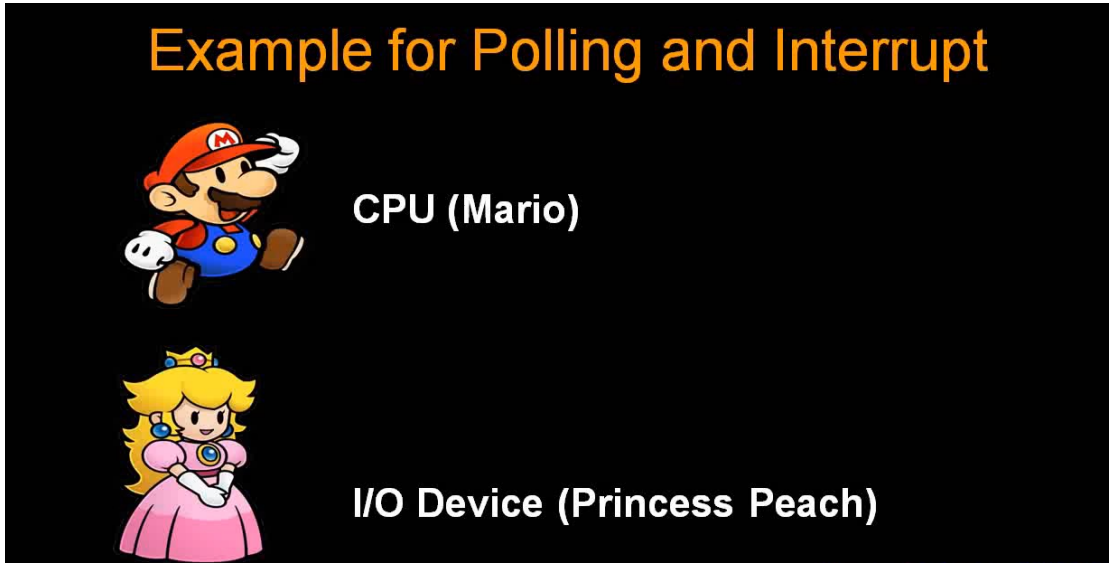
Polling



Interrupt



ISR



- An interrupt in a microprocessor is a signal that requests the microprocessor's attention. It can be generated by either an external device or by the microprocessor itself. When an interrupt occurs, the microprocessor suspends the execution of its current program (temporarily) and jumps to a special program called an interrupt service routine (ISR). The ISR handles the interrupt and then returns control to the main program.
- Interrupts can be classified into two main types: hardware interrupts and software interrupts.
  - Hardware interrupts are generated by external devices.
  - Software interrupts are generated by the microprocessor itself, such as when an instruction is executed that causes an error.

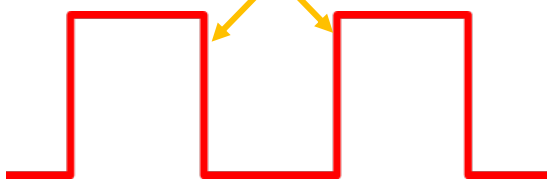
Interrupt in the Lab code!

```
attachInterrupt(digitalPinToInterrupt(channelA), updateEncoderStateMachine, CHANGE);
attachInterrupt(digitalPinToInterrupt(channelB), updateEncoderStateMachine, CHANGE);
```

```
/* Encoder input pins (used for state machine and interrupts) */
#define channelA 2
#define channelB 3
```

```
/* Set encoder pins as input but with pullup resistors*/
pinMode(channelA, INPUT_PULLUP);
pinMode(channelB, INPUT_PULLUP);
```

```
void updateEncoderStateMachine()
/* User written code to update state and increment count of state machine  */
{
  channelAState = digitalRead(channelA);
  channelBState = digitalRead(channelB);

  switch (state)
  {
     case state1:
     if (channelAState && !channelBState)
     {
        count++;
        state = state2;
     }
     /* else if .... a lot of code goes here! */
     /* don't forget "break" at end of each case. */
  }
}
```



Summary

- Concept of simple state table introduced
- Concept of state diagram introduced
- Finite state machine
- State machine using 2D state table
- State machine based on case statement
- Simple examples:
  - Traffic lights
  - Traffic lights with car/pedestrian crossing
- Introduction to Lab 1
- Interrupt