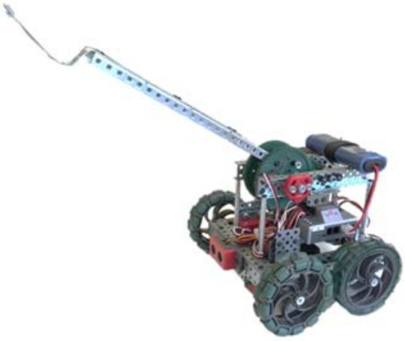
MECHENG 201

Introduction to Microcontrollers, Robot Programming, and Control Systems



Dr. Hazim Namik
h.namik@auckland.ac.nz

VEX Lab Schedule

- Lab location: 403 – 319 (Eng 3.319)
- Stick to your enrolled session (know your lab group)

Week	Lab Group A	Lab Group B	Lab Group C	Lab Group D	Lab Group E
5-12	Wed 8-10am	Thursday 8-10am	Thursday 11am-1pm	Friday 8-10am	Friday 12-2pm

- First Lab:
  - Before coming to the lab, READ sections 1 – 3 of the lab manual
  - Find a lab and project partner **in the same lab group**
  - Expectation: reach Task 5.
- **Limited** contact time with robot
  - Need to work on code outside lab times and only test and modify algorithms during scheduled labs for maximum benefit.

Overview

Introduction to VEX Robots

C Programming & Flowcharts

Advanced VEX Programming

Lab (5%)

PROJECT (15%)

Introduction to Control Systems

Control Loops on the VEX

Introduction to Microcontrollers

Table of Contents

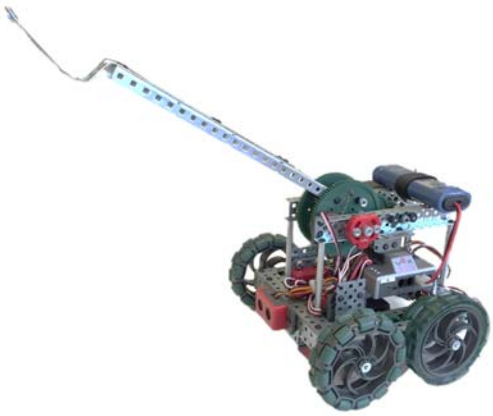
MODULE A

Programming Robots

1. VEX Robot Introduction
2. C Programming & Flowcharts
3. Advanced VEX Programming

PART A-1

VEX Robot Introduction



What is a Robot?

- The term “robot” has evolved (and continues to) over the years.
- Modern definition: A robot is a system that
  - physically exists in the real world;
  - can sense its environment or part of it;
  - can act based sensor measurements to achieve an objective; and
  - is autonomous (not remotely controlled by an operator)
- Still a very broad definition but let’s not dwell on it.

Question

- According to the definition, which of these are robots?
  - Smartphone
  - Heat pump
  - Automatic door
  - Roomba
  - Boeing 777

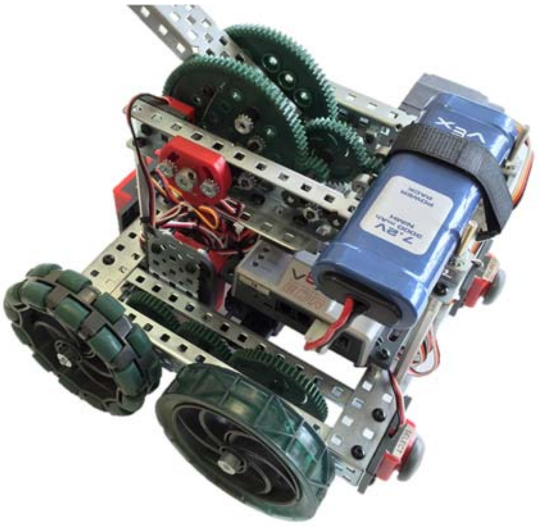
What Makes a Robot?

1. Structure/body
   - A frame or chassis to the other components together
2. Sensors
   - Devices that measure or detect certain attributes of the environment or the robot itself
   - Examples: buttons, limit switches, shaft encoders, temperature, light, vision, magnetic, lasers, sonar, etc.
3. Actuators
   - Devices that manipulate/affect their environment
   - Examples: motors, hydraulic or pneumatic cylinders, etc.

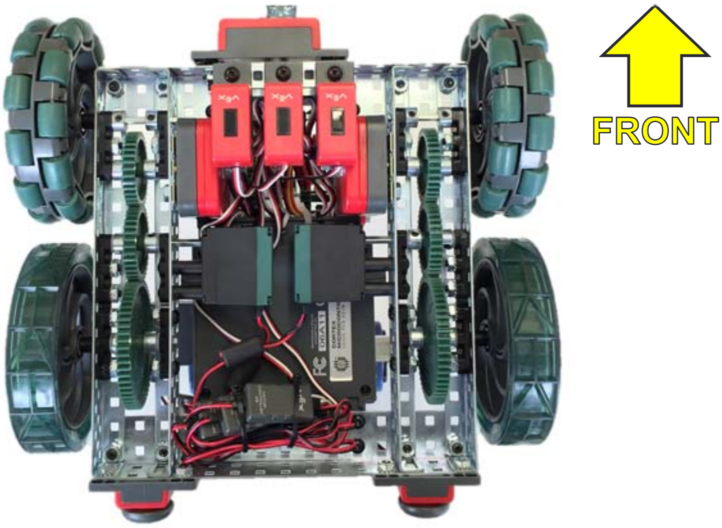
What Makes a Robot?

4. Computer controller(s)
   - Decides the sequence of actions to take (i.e. command the actuators) based on sensor measurements to achieve its objective.
   - Can be a cheap microcontroller or expensive processing system.
     - Depends on processing power required.
5. Power source
   - The source of energy to power the *active* sensors and actuators as well as the computer controller.
   - Usually electrical power stored in a battery for mobility.

Hardware



Hardware



Software - RobotC

- RobotC is a special version of C
  - Some C features are not available
    - E.g. `double` data type
  - Special and custom features built in for the VEX robot
    - E.g. `motor[]`, `SensorValue[]` commands
  - Special RobotC **I**ntegrated **D**evelopment **E**nvironment (IDE)
    - *RobotC for VEX Robotics (version 4.X)*

- Entry point into the program is

```
task main (){
   // code goes here
   }
```

instead of

RobotC Hardware Interface

- Need to tell the program how the microcontroller is connected to the sensors and actuators

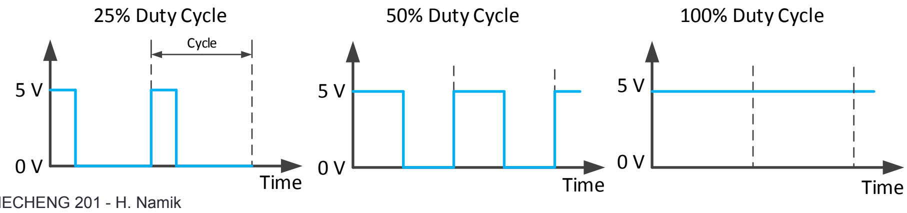
- Can be done semi-automatically in RobotC IDE
  - You define what type of sensor/actuator is connected to which port and the RobotC code is automatically generated
  - THIS HAS BEEN DONE FOR YOU

- Template files available on Canvas
  - Has special functions to ***safely*** control the arm → use them!
  - Use it for the labs and project as a starting point

Hardware Interface – Output Ports

Actuator	Signal Type	Output Port
Arm motor	PWM	Output port 2
Right wheel motor	PWM	Output port 7
Left wheel motor	PWM	Output port 8

- PWM: Pulse Width Modulation
  - Used to change the applied voltage to the DC motor by rapidly switching it on and off.



Hardware Interface – Input Ports

Sensor	Signal Type	Input Port
Left light sensor	Analog	Analog input 1
Middle light sensor	Analog	Analog input 2
Right light sensor	Analog	Analog input 3
‘Stop’ button	Digital	Digital input 1
‘Start’ button	Digital	Digital input 2
Left wheel encoder	Digital	Digital inputs 5 & 6
Right wheel encoder	Digital	Digital inputs 7 & 8
Arm encoder	Digital	I2C serial comm
Arm low-limit switch	Digital	Digital input 11
Arm high-limit switch	Digital	Digital input 12
Sonar	Digital	Digital input 9

VEX Robot Input/Output Setup

- Automatically generated I/O setup

```
#pragma config(Sensor, in1,    lightLeft,      sensorReflection)
#pragma config(Sensor, in2,    lightMid,       sensorReflection)
#pragma config(Sensor, in3,    lightRight,     sensorReflection)
#pragma config(Sensor, dgtl1,  btnStop,        sensorTouch)
#pragma config(Sensor, dgtl2,  btnStart,       sensorTouch)
#pragma config(Sensor, dgtl5,  encRight,       sensorQuadEncoder)
#pragma config(Sensor, dgtl7,  encLeft,        sensorQuadEncoder)
#pragma config(Sensor, dgtl9,  sonar,          sensorSONAR_mm)
#pragma config(Sensor, dgtl11, armLimit_low,   sensorTouch)
#pragma config(Sensor, dgtl12, armLimit_high,  sensorTouch)
#pragma config(Sensor, I2C_1,  armEncoder,     sensorQuadEncoderOnI2CPort,
, AutoAssign )
#pragma config(Motor,  port2,  motorArm,       tmotorVex269_MC29, openLoop,
reversed, encoderPort, I2C_1)
#pragma config(Motor,  port7,  motorRight,     tmotorVex269_MC29, openLoop)
#pragma config(Motor,  port8,  motorLeft,      tmotorVex269_MC29, openLoop,
reversed)
```

Unique name

Motors

- Syntax:

```
motor [ motorName ] = power ;
```

  - `motorName` → unique motor name
  - `power` → integer power level ranging from -127 to 127

- IMPORTANT:
  Motor will stay at the specified `power` level until the next `motor[]` command.
  - Setting motor power to zero will stop the motor.

- Each motor may drive differently at the same power level!

Motion and Motor Commands

Motion	Left Motor	Right Motor
Drive forward		
Reverse		
Right turn (on the spot) – clockwise		
Left turn (on the spot) – anticlockwise		

- General syntax:

```
motor [motorLeft] = power;
motor [motorRight] = power;
Wait1Msec(time);
motor [motorLeft] = 0;
motor [motorRight] = 0;
```


Provided Arm Functions

Function	Description
`armUp(float pctPwr);`	Raises the arm to its highest position (until it hits the upper limit switch) at the given *percentage* power level.
`armDown(float pctPwr);`	Lowers the arm to its lowest position (until it hits the lower limit switch) at the given *percentage* power level.

- Parallel arm task:
  - In the `main()` task, you'll see
  - The `checkArm` task ensures the safety of the arm.
  - Do **NOT** insert any code before that line!

```
task main() {
   startTask(checkArm);
   startTask(checkButtons);
   .
   .
   .
}
```

Sensors

- Syntax:

```
result = SensorValue [ sensorName ];
```

  - `result` → *predefined* `int` to store sensor measurement
  - `sensorName` → unique predefined sensor name

- Sensors

Sensor	Data Range	Description
Touch (buttons)	0 to 1	0 = not pressed, 1 = pressed
Light	0 to 4095	Value retuned depends on ground colour.
Potentiometer	0 to 4095	Value retuned depends on arm position
Encoder	-32768 to 32767	Every 360 counts corresponds to 1 encoder shaft revolution

Light Sensors

- Light sensors are used to detect ground colour changes
  - Can be utilised to follow lines or detect borders

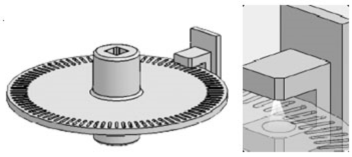
- The returned reading for the same colour can be
  - different for each sensor
  - affected by external light sources
  - varying across a range of values (i.e. not constant)

- Use the debugging tools in the RobotC IDE to check the reading for each sensor on specific colours (black, white, etc.)

Optical Shaft Encoders

- The optical encoder contains a slotted wheel and a light sensor.
  - As the shaft rotates, the light sensor inside generates a digital signal that correspond to counts per revolution
- Quadrature encoders have two light sensors to detect direction of rotation.
- VEX encoders measure rotation of the shaft since power up or last reset
  - Cannot determine absolute position
- To reset encoder

```
SensorValue[encName]=0;
```



Source: http://www.vexrobotics.com/wiki/images/2/22/Optical_Shaft_Encoder_Figure_2.jpg

Wheel Encoders

- Wheel encoders can be used to work out the linear distance travelled or the angle turned by the robot *assuming no slip*.
- To convert between encoder counts to linear distance, use the following information:
  - Every 360 encoder counts equates to a complete encoder shaft revolution.
  - Due to the gear ratio, every 3 wheel revolutions correspond to 5 encoder shaft revolutions.
  - The wheel diameter is 103mm.

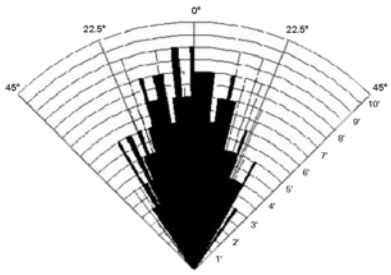
$$\therefore \text{Distance (mm)} = count$$

Arm Encoder

- The arm encoder can be used to determine the absolute position of the arm
  - First you must move the arm to one of the known limits (upper or lower) which we know/can measure the real arm angle
  - Once the arm is there, reset the encoders once.
  - Each robot will have it's unique values for the minimum and maximum arm positions.
- To convert between arm encoder counts to rotated arm angle, use the following information:
  - Every revolution of the encoder shaft produces 240.448 counts.
  - The combined gear ration between the arm and the encoder shaft is 1:21 (i.e. 1 revolution of the arm corresponds to 21 revolutions of the encoder shaft).

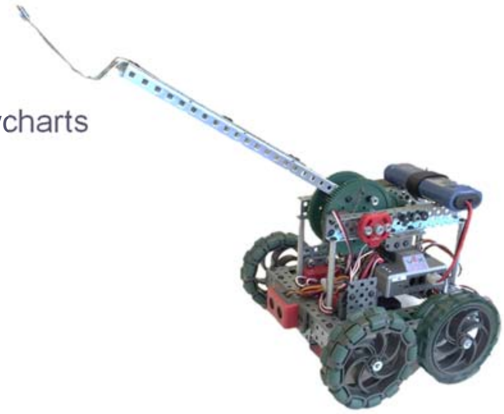
Sonar

- Sends 40kHz ultrasonic pulses and measures the time it takes for the reflected waves to arrive to calculate the distance.
  - Assumes a constant speed of sound
  - Some surfaces may absorb these ultrasonic waves
  - Has an apporx. 30° beam angle
- Use it to detect obstacles
  - Range 30 – 3000 mm
  - Sensor returns -1 if no object is detected.
- RVW Demo. Take notes

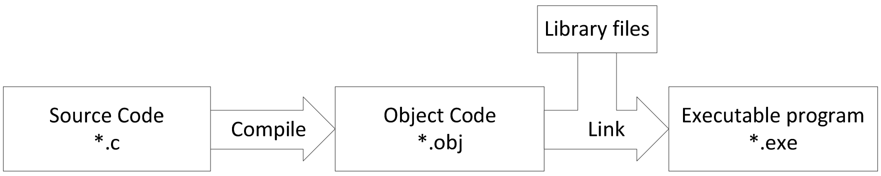


PART A-2

C Programming & Flowcharts



Programming in C



- Source code can be written in any text editor.
  - Some have advanced features to make programming easier (known as IDE – integrated development environment)
- The code is then compiled by the *compiler* and then linked with standard C libraries by the *linker*.
- The executable file does not require the source or object code to run.

C Program Components

```
#include <filename.h>

int a, b, c;   // This is single line comment

double someFunction (int x, double y);

/* Multi-line comments
are possible */

int main (void) {
    int d;
    double  e, f;
    e = someFunction (a,f);
    return 0;
}

double someFunction (int x, double y){
    // Some code goes here
}
```


Functions

- A Function is a section of code that
  - is named;
  - is independent;
  - performs a specific task; and
  - can return a value to the calling program.
- The function doesn't execute unless it is *called* by the main program.
- Allow you to structure your program into functional blocks
  - Program becomes easier to write
  - Easier to debug

Functions

- Each functions must have a *prototype* defined *before the main function*
  - Syntax:
```
return_type function_name (arg_type name_1, ...,
arg_type name_n);
```

- Function *definition* can be placed *after the main* function
  - Syntax:
```
return_type function_name (arg_type name_1, ...,
arg_type name_n){
    /* statements */
}
```


Example

- This program asks the user for two integers and returns their average.

```
#include <stdio.h>

float getAverage(int num1, int num2);

int main (void){
    int number1, number2; float average = 0;

    printf("Please enter two integers: ");
    scanf("%d %d",&number1, &number2);

    average = getAverage(number1, number2);
    printf("\nAverage = %2.1f \n\n",average);
}

float getAverage(int num1, int num2){
    float average = (num1 + num2)/2;
    return average;
}
```

Programming Errors

Syntax Errors

- Caused by unknown commands or illegal syntax usage.
- Usually arise from typos or not knowing the correct usage
- Compiler will throw and error and (hopefully) give useful information about the error

Logical Errors

- Caused by incorrect implementation of an equation or algorithm
  - not easy to find
- Code compiles and may even run but results are wrong
- Build your code in modules and create test cases to verify result/output

Programming Errors - Example

- Find and correct all errors in the following code excerpt that calculates the roots of a quadratic function.

```
int a, b, c, x1, x2;
a = 3.5; b = -2; c = -1;
x1 = -b+sqrt[b*b-4*a*c]/2*a
X2 = -b-sqrt[b*b-4*a*c]/2*a
```

- Correct version:

Pseudocode

- Informal (syntax-wise) way of describing the procedure/algorithm that is easy to read and understand.

```
for (i = 1; i<=100; i++) {
    set print_number to true;
    if i is divisible by 3
        print "Fizz ";
        set print_number to false;
    if i is divisible by 5
        print "Buzz";
        set print_number to false;
    if print_number, print i;
    print a newline;
}
```

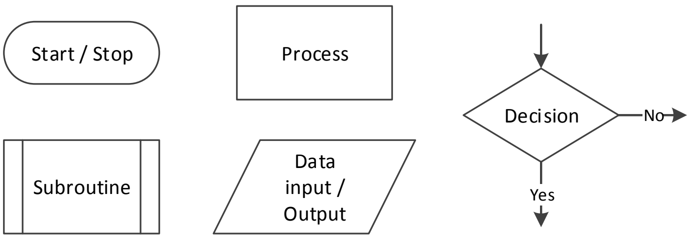

Fizz-Buzz C-code Example

- Recall the pseudocode example?

```
#include <stdio.h>
int main (void){
   for (int i = 1; i<=100; i++) {
      int print_number = 1;
      if (i%3==0){
         printf("Fizz "); print_number = 0;
      }
      if (i%5==0){
         printf("Buzz"); print_number = 0;
      }
      if (print_number){ printf("%d", i);}
      printf("\n");
   }
   return 0;
}
```

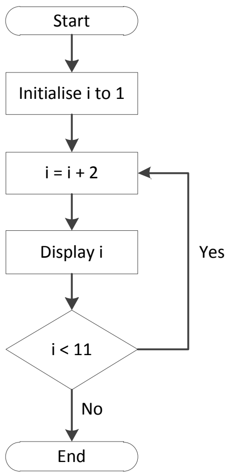
Flow Charts

- Used to visually describe the program flow.
- Do NOT write code in flowchart symbols
- Set of standard symbols:



Example

- What numbers will be displayed by this program?



If Statement Flow Chart

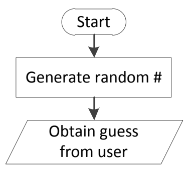
- Draw the flow chart of the code segment below

```
if (expression){
   statement_1;
   statement_2;
}
else{
   statement_3;
   statement_4;
}
statement_x;
```


Guessing Game

- Generate a random integer between 0 and 100
- Ask the user to guess it.
- Provide a “higher” or “lower” clue based on last guess.
- Keep asking until correct number is guessed.

- **On a new page/refill:**
  - **Complete flow chart**
  - **Complete C code**



```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main (void){
     srand (time(NULL));// generates a new random seed everytime
     int secretNumber = rand() %100;// generate random in between 0 and 100



          printf("Please enter an integer between 0 and 100: ");
          scanf("%d",&guess);
```

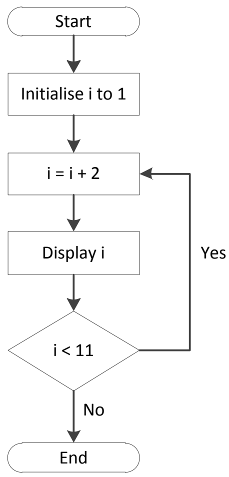

Homework A-2

- Draw the flow chart for the pseudocode below

```
for (i = 1; i<=100; i++) {
     set print_number to true;
     if i is divisible by 3
          print "Fizz ";
          set print_number to false;
     if i is divisible by 5
          print "Buzz";
          set print_number to false;
     if print_number, print i;
     print a newline;
}
```

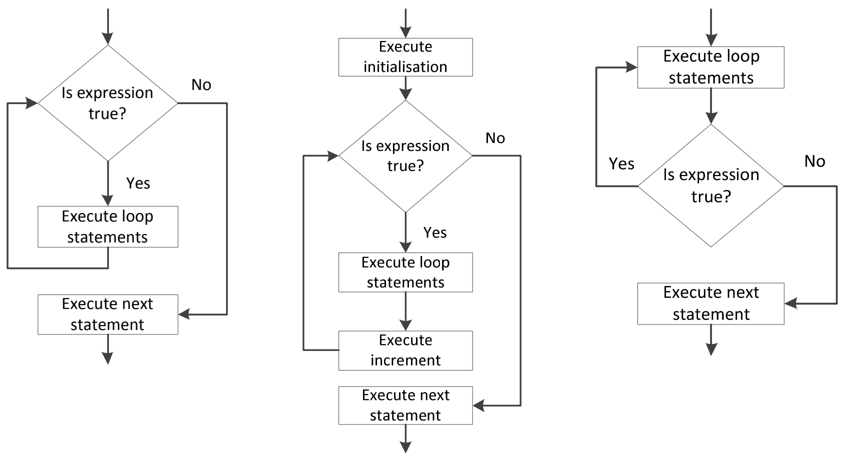
Homework A-2

- Convert the flowchart into C code



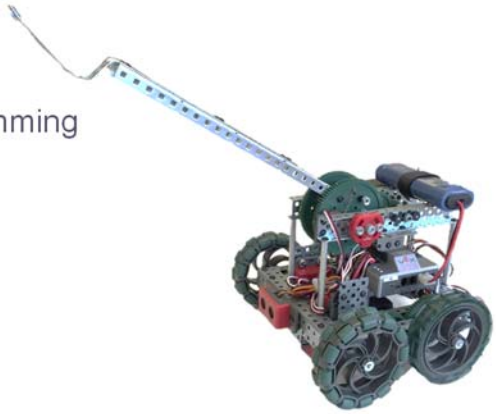
Homework A-2

- Identify the type of programming loop for these flow charts:



PART A-3

Advanced VEX Programming



Timers

- Timers are used to measure elapsed time of certain events
  - They run in the background (wait commands pause the program)
  - Timers use signed 32-bit integers
  - Tick every 1 millisecond
  - Start counting when the program starts
  - Should be reset before overflowing
- How many hours can the timers run before overflowing?

Using Timers

- Cortex microcontroller has 4 timers named T1 to T4
  - T3 is used for background button checking → Do **NOT** use T3

- Timer-related commands:

Command	Descritpion
`ClearTimer(timerName);`	Resets the named timer (`T1` to `T4`) to zero
`time1[timerName];`	Returns the specified timer count (integer) in milliseconds units

Example

- Write a program that drives forward for 5 seconds unless the Stop button is pressed to stop.

```
task main() {
        startTask(checkArm);        // DO NOT DELETE THIS LINE
        // -------------------- Put your own algorithms here -----------









        // -----------------------------------------------------------
        stopAllTasks();    // end of program - stop everything
}
```

Basic Multitasking

- Program starts by running the `main()` task
- The Cortex microcontroller can run up to 19 additional tasks
  - Without knowing the advanced multitasking commands, do not start more than 4 concurrent tasks excluding the ones in the template.
- Commands

Command	Description
`StartTask(taskName);`	Starts running the designated task
`StopTask (taskName);`	Stops executing the designated task
`StopAllTasks();`	Stops all running tasks including `main()`

Declaring a task

- Syntax:

-

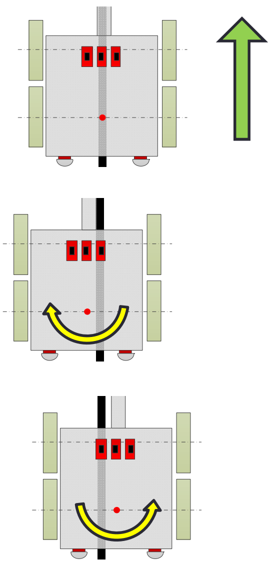
- Tasks have to be defined before they are started
- Like functions, tasks have prototypes and their own scope
- Unlike functions, tasks do not have inputs or return arguments

Demo

- Take your own notes!

Line Following

- Basic idea:
  - Keep the middle sensor on the line
  - Use right and left light sensor to turn
    - Turn for how long?
- Other considerations
  - Light sensors **must** be in front of driving wheels
  - Driving and turning speeds
  - Detect black, white, not black or not white?

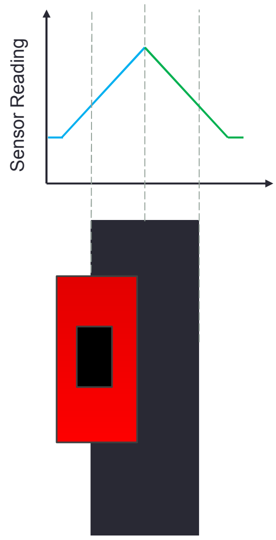


Basic Line following

- Basic logic
  - 3 line sensors
  - 2 possible outcomes from each light sensor (black and not black)
  - $2^3 = 8$ possible combinations
  - Need to define an action for every possible situation that will ensure line following
- Enhanced version - untested ideas
  - Place the turning point on the line before turning to align with the line – cornering issue?
  - Drive straight (not distance) using encoders while the mid light sensor is on black – useless if not aligned with black line
  - Remember which sensor detected black last and act as soon as mid sensor loses the black line – cornering issue?

Advanced Line Following

- Edge following
  - Follow the edge of a line
    - Know what the edge of a line should read
    - Use the inside edge
  - Use a P or PI controller to track that sensor reading keeping the middle sensor on the black line
    - Controller commands change in base power to each wheel to turn
  - Controller is stable for one edge and unstable for the other
    - Need additional logic if the mid sensor loses the black line or when it hits a corner
  - Robot needs to move slowly



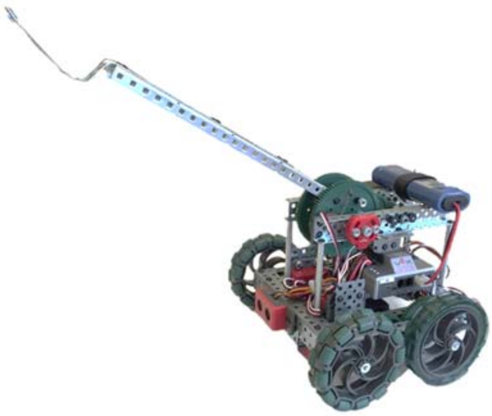
MODULE B

Control Systems

1. Introduction to Control Systems
2. Implementing Control Loops on The VEX Robot

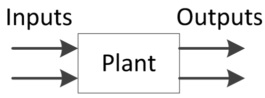
PART B-1

Introduction to Control Systems



Introduction to Control Systems

- Terminology
  - Plant: the system we are trying to control



  - Actuators: devices that are able to manipulate the plant's inputs to influence the plant's outputs
  - Sensors: measure outputs/signals from the plant
  - Controller: commands the actuators to manipulate the plant output based on sensor measurements to achieve a desired output

Block Diagrams

- Block diagrams represent a single process or system dynamics

- Mathematically, can be expressed as an algebraic or a differential equation
  - Usually represented as a transfer function $T(s)$ in the Laplace domain as

Open Loop (OL) Control

- Objective: Reference command = Plant output
- Block diagram:

- Features:
  - Does not use sensors to measure output
  - Actuator commands are based on pre-determined function of the reference command
  - Sensitive to disturbances or changes in the plant properties
- Examples:
  - Toasters, time-based dryers, dishwashers, microwave oven, bowling

Exercise – Complete the Table

Example	Plant	Actuator(s)	Sensor(s)	Controller
Toaster				
Time-based dryer				
Dishwasher				
Microwave				
Bowling				

Closed Loop (CL) Control

- Objective: Reference command = Plant output
- Block diagram:

- Features
  - Uses sensors to measure output (feedback control)
  - Actuators are commanded based on the error (reference – output) to drive the error to zero.
  - Robust against disturbances and changes in plant properties
  - More complex than open loop control

Closed Loop Control

- Examples
  1. Thermostats in heaters, ovens, refrigerators, etc.
  2. Water level in tanks (e.g. toilet cistern)
  3. Auto-pilot on commercial planes
  4. Anti-lock Braking System (ABS) in modern cars
  5. Clothes dryer with humidity sensor
  6. Mouse cursor position on the screen (human controller)
  7. Standing up

Homework – Complete the Table

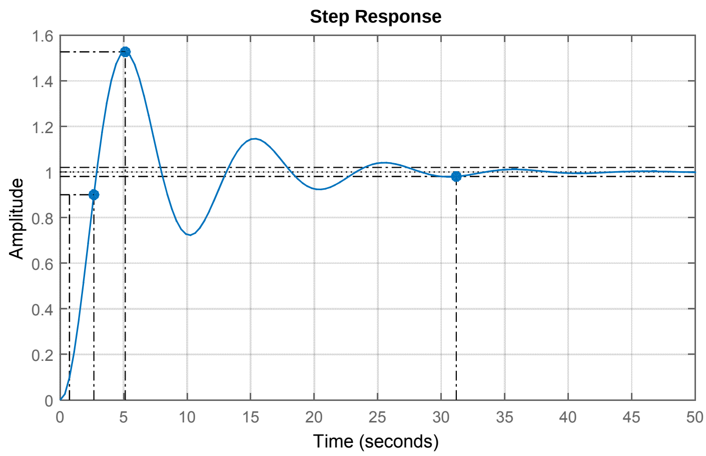
Example	Plant	Actuator(s)	Sensor(s)	Controller
1 (oven)				
2				
3 (altitude control)				
4				
5				
6				
7				

CL Performance Measures Definitions

- Steady-state value
  - The final value of the plant output (as t → ∞)
- Rise Time
  - The time it takes for the system to go from 10% to 90% of the steady-state value
- Overshoot
  - The difference between the peak value and the steady-state value
- Settling Time
  - The time when the response is within 2% of the steady-state value
- Steady-State Error
  - The difference between the desired value (reference) and the steady-state value

CL Performance Measures

- Response is measured to a step input



Closed Loop Controllers

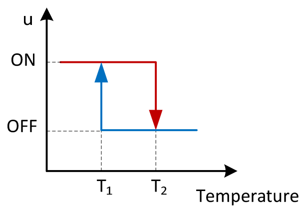
- Feedback controllers can be but not limited to
  - Basic: a simple and straight forward controller
  - Robust: a controller that maintains performance despite changes in environment, plant properties, etc.
  - Adaptive: a controller that adjusts its "behaviour" to maintain performance
  - Predictive: a controller that looks ahead and predicts the plant output and adjusts its commands accordingly

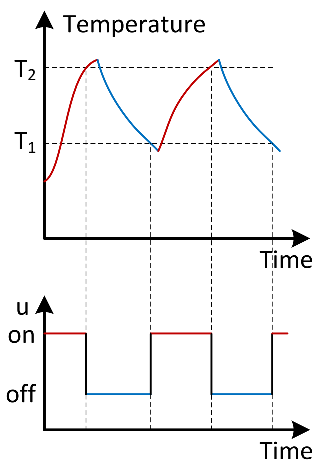
- We will look at the following basic controllers
  - On-off controller
  - Proportional controller (P controller)
  - Proportional-Integral controller (PI controller)

On-off Control

- Simplest feedback controller
  - Control action, *u*, can either be ON or OFF
  - Hysteresis zone to avoid frequent switching
  - Used when precise control is not necessary

- Best example: Thermostat



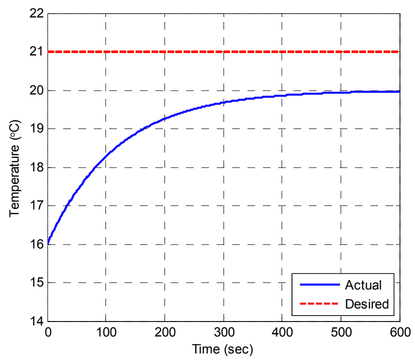


Proportional Control

- Control action, *u(t)*, depends linearly on the error and a proportional gain K_p
- The further away the plant output *y(t)* from the reference point *r(t)*, the bigger the control action.
- Smooth motor response
- May have a steady-state error (does not reach the desired reference value) depending on the plant type.

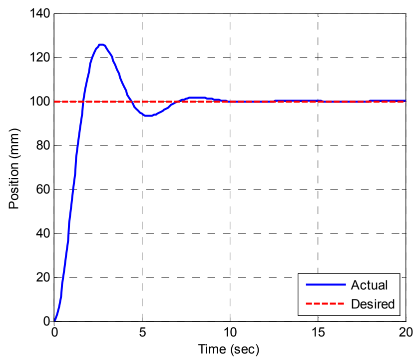
P Control Examples

Temperature Control



Type 0 system

Position Control



Type 1 system

Proportional-Integral Control

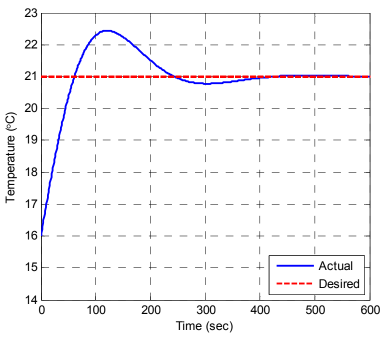
- A P controller with an additional term that integrates the error

- The integral action command is proportional (K_i) to the integral of the error signal
  - This eliminates steady state errors in most cases

- Adding the integral action can also
  - Slow down the system (depends on tuning the gains)
  - Reduce the stability of the plant

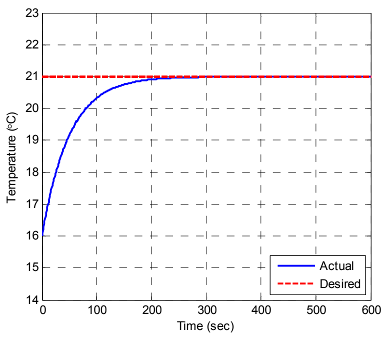
PI Control Example - Temperature

Bad Tuning



Type 0 system

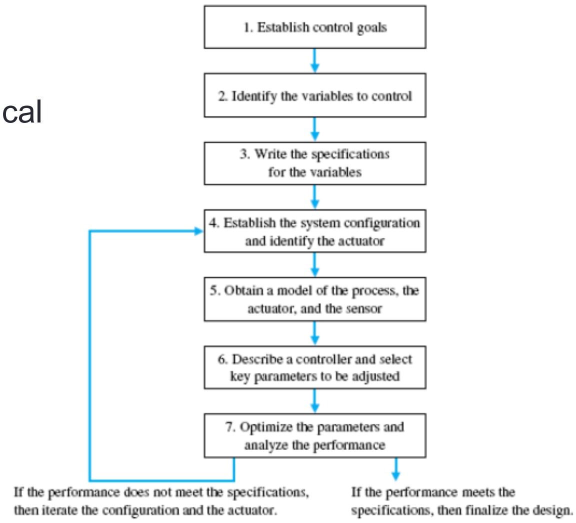
Good Tuning



Type 0 system

Control Design Process

- Model Based design
- Require a mathematical model of plant for control design
  - Outside scope of this course
- Iterative process



Source: Dorf, R. and Bishop, R. "Modern Control Systems".

Example (Open Loop)

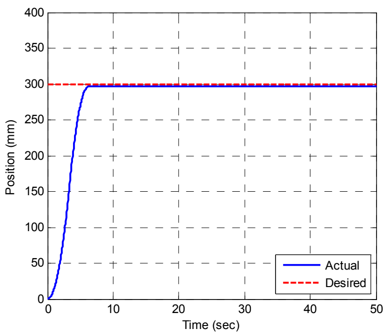
Design an open loop controller to drive the VEX robot x metres forward

- What actuators do we have?
- What can we control?
- Do we need sensors?
- What is the relationship between reference command and plant output?

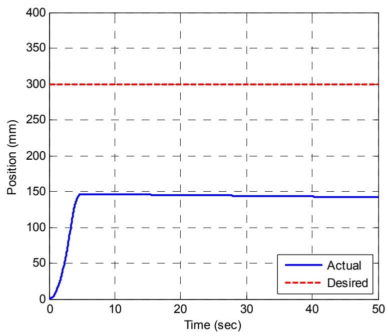
What happens when robot is going up an incline?

Example (Open Loop)

Flat Surface



30° Incline



Example (Closed Loop)

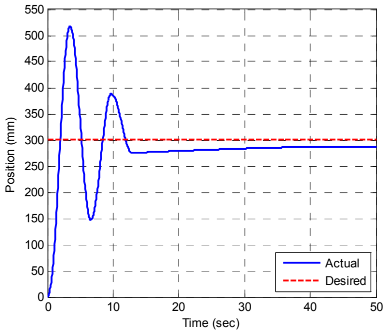
Design a closed loop controller to drive the VEX robot x metres forward

- What actuators do we have?
- What can we control?
- What sensors do we have?
- What is the relationship between reference command and plant output?

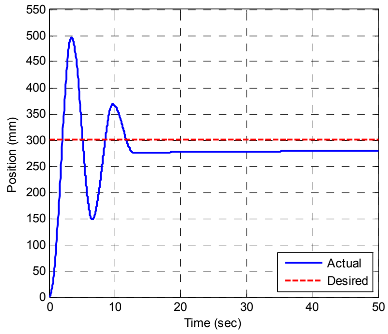
What happens when robot is going up an incline?

Example (P-Control)

Flat Surface

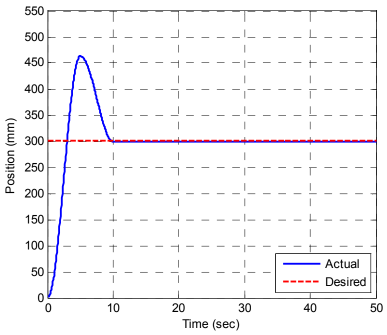


30° Incline

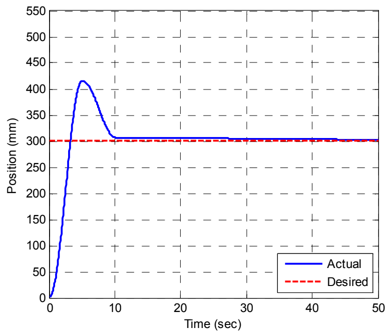


Example (PI-Control)

Flat Surface



30° Incline



Effects of Controller Gains

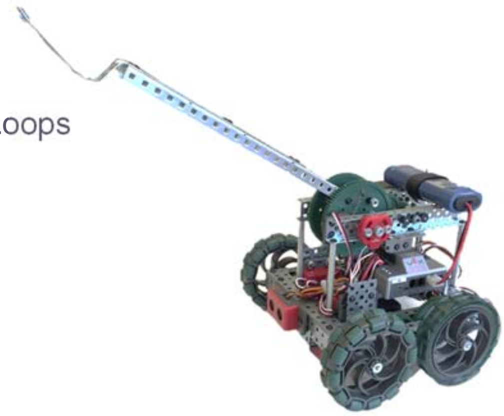
- Effects of increasing the controller gains on the system performance are summarised in the table below

	Rise Time	Overshoot	Settling Time	Steady-state Error
K_p				
K_i				

- Note: these are general indicators of what to expect when the controller gains are increased.

PART B-2

Implementing Control Loops on the VEX Robot



Implementing a Control System on the VEX Robot

- Normally, a control loop is ALWAYS running (i.e. infinite loop)
- However, for the VEX robot, the program must proceed to the next set of instructions once the goal is reached
  - Can have parallel control loops running indefinitely
  - But some control loops may never reach their goal (e_{ss})
- Therefore, we must define some conditions that would terminate the control loop

Example: Driving a Set Distance

- Use a P-controller to drive a certain distance? Draw the block diagram.

Example: Driving a Set Distance

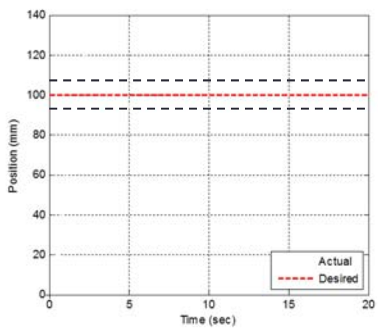
- Use a P-controller to drive a certain distance? Write the code.

```
void drivePcont(float target, float Kp){
   // initialise variables
   float  currentPos, error, u=0;
   int Pwr = 0;
   do{
      currentPos = ... // get current position from wheel encoders
      error = target - currentPos;   //update the error (E=R-Y)
      u = Kp*error;   // calculate the control effort
      Pwr = (int) saturate(u,-50,50); // saturate motor power
      // send power to motors
      motor[motorLeft]  = Pwr ;
      motor[motorRight] = Pwr ;
      wait1Msec(100);             // repeat at ~10Hz
   } while(1);
}
```


Terminating a Control Loop

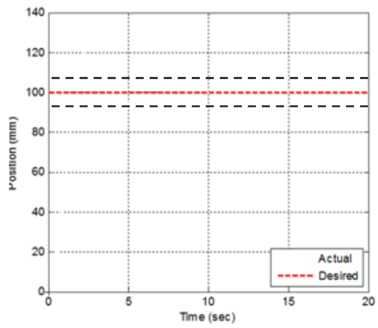
- **Tolerance Based Condition(s)**
  - Exit the control loop when the error is below a predefined tolerance (e.g. stop if within ±5 mm)

- Limitation(s):



Terminating a Control Loop

- **Change in Error Condition**
  - Exit the control loop when the change in error (or rate of change of error) is zero or within a small tolerance (i.e. when the output reached a steady-state value)
  - This addresses the issue of steady-state error
- Limitation(s):



Terminating a Control Loop

- **Timer Based Conditions**
  - Exit the control loop when
    - Either error is within the specified tolerance OR the change in error is small; AND
    - The above condition remains TRUE for a specified time period (e.g. 2 seconds)
- Limitation(s):

MODULE AB

Addressing Control and Robot Issues

Robot Issue: Driving Straight

- Issue:

- Possible solution or mitigation:

Example: Driving Straight

- How can we use a control system to ensure that the robot is driving straight? Draw the block diagram.

Robot Issue: Acceleration

- Issue:

- Possible solution or mitigation:

Robot Issue: Backlash

- Issue:

- Possible solution or mitigation:

Robot Issue: Slip

- Issue:

- Possible solution or mitigation:

Control Issue: Steady-state error

- Issue:

- Possible solution or mitigation:

Control Issue: Integrator Windup

- When does integrator windup occur?
  - Integrator windup occurs when an actuator is **saturated** in the presence of a controller with an **integrator component** (e.g. PI controller)

- How does it happen?
  - When actuator is saturated, the plant responds slower

    → The integral of error with time increases

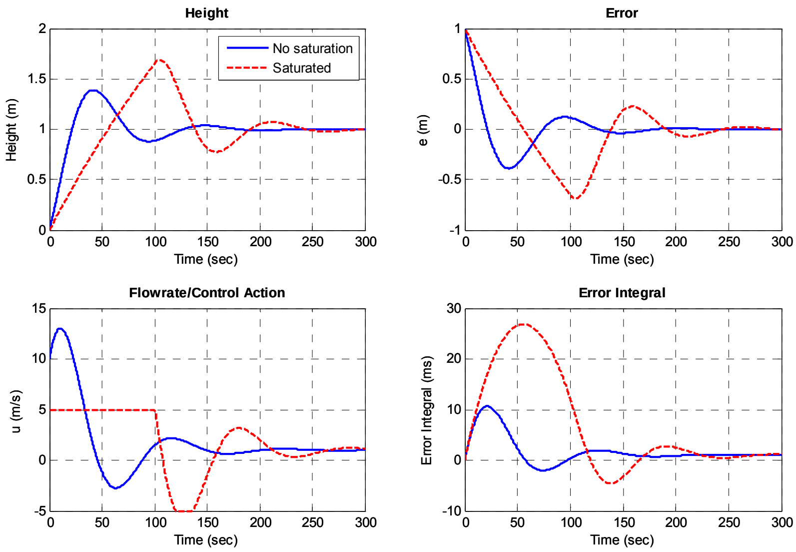
    → This increases the integral action $\left(K_i \int edt\right)$ but that increase does nothing since actuator is saturated

    → Area keeps getting bigger until the output overshoots the desired value

    → Controller cannot reverse action until area is small or zero

    → Cycle repeats until error is small enough not to cause saturation

Kp = 10 and Ki = 1

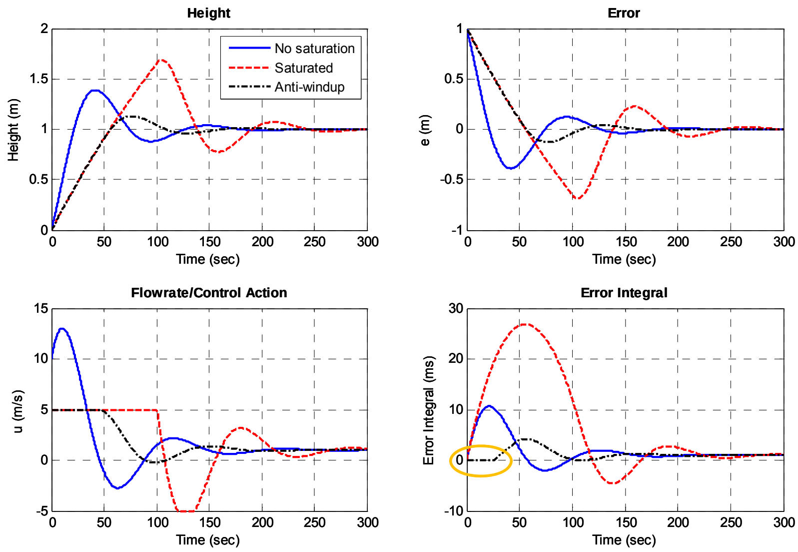


Control Issue: Integrator Windup

- Why is integrator windup an issue?

- How to overcome the integrator windup issue?
  - Reduce controller gains to avoid saturation
    - Slows the responsiveness of the system
  - Many anti-windup schemes available

Kp = 10 and Ki = 1



MODULE C

Introduction to Microcontrollers

Different Numbering Systems

- Decimal system is base 10
  - Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - Used almost everywhere (mathematics, finance, etc.).
  - Not suitable for computers
- Binary system is base 2
  - Digits: 0, 1
  - Digital logic is either ON of OFF
  - Suitable for computers
- Hexadecimal system is base 16
  - Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
  - Suitable for computers with large memory addresses

Numbering Systems

Decimal	Hexadecimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Binary Numbers and Computers

- Computers have limited number of bits available
- An n-bit number has an integer range:
  - Unsigned : from 0 to $2^n - 1$
  - Signed : from -2^{n-1} to $2^{n-1} - 1$
- Example:
  - With an 8-bit system, integers can range:
    - Unsigned :
    - Signed :

Negative Numbers

- How do you represent negative numbers in binary?
- One method is called two's complement :
  - Main benefit: Can use existing binary adders without any modifications
  - Result is extremely dependent on the number of bits (i.e. the two's complement of -2 in 3-bits is not the same in 4-bits)
- Method:
  - Step 1: Take the complement of the binary number (invert the bits – 1 becomes 0 and 0 becomes 1)
  - Step 2: Add 1

Two's Complement- Example

- Express -3 in two's complement using 3 bits

- Add 1 to -3:

	Binary	Decimal
	001	1
+		-3
=		

Overflow

- What happens when we add 2 + 3?

```
   010
+  011
=
```

- Two +ve numbers added and the result is negative!
  - The adder returns an overflow flag or error indicating this operation cannot be done on this system.

Binary	Decimal
000	0
001	1
010	2
011	3
100	-4
101	-3
110	-2
111	-1

3-bit two's complement table

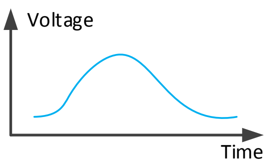
Bits and Computers

- The number of bits for a processor determines the precision and maximum number of addressable memory locations
  - Common configurations: 8-bit, 16-bit, 32-bit, 64-bit
- 32-bit computers:
  - A 32-bit processor can access 2^{32} = 4,294,967,296 memory addresses
  - Normally, each memory address is one Byte
  - Therefore, maximum accessible memory is 4GB
- How many Terra Bytes (TB) can a 64-bit computer access?

Analog and digital signals

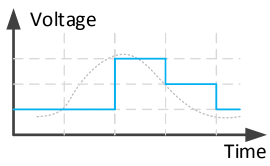
Analog Signals

- Continuous in time and magnitude



Digital Signals

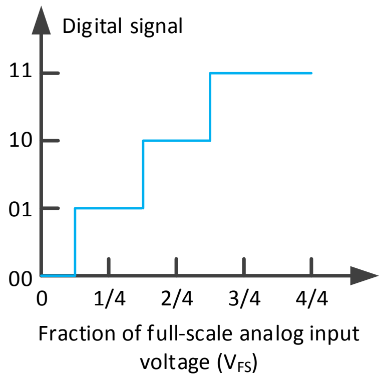
- Discrete in time and magnitude



Analog to Digital Converter (ADC)

- Readings from analog sensors need to be converted to a digital signal for the microcontroller to understand.

- For an n-bit ADC, there are 2^n quantisation levels
- The resolution (minimum input voltage change to cause a change in digital output) is $V_{FS}/2^n$
- Maximum quantisation error is $V_{FS}/2^{n+1}$



ADC Example

- For an 8-bit ADC and an input voltage range of 0-5V. Find the resolution and maximum quantisation error.

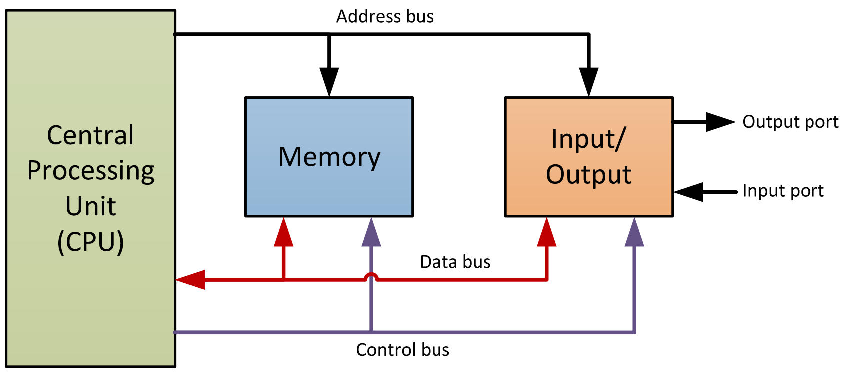
Homework C

- A linear temperature sensor outputs 30 mV/°C. If the maximum operational temperature range is 100°C, determine the number of bit required for the ADC to measure the sensor input with a resolution of 0.5°C.

Why do I need to know this?

- Know how numbers are stored and represented inside a computer.
- Know the limits of your system in terms of number ranges and memory size.
- Useful to control input/output ports of microcontrollers.

What is a Microprocessor System?



Components of a Microprocessor System

- Microprocessor or Central Processing Unit (CPU)
  - Executes program instructions.
  - Has three basic elements:
    1. Control unit: Determines timing and sequence of operations.
    2. Arithmetic and logic unit (ALU): Data processing (e.g. adding two numbers).
    3. Registers: Internal memory locations to store temporary results while the ALU is processing instructions.
- Buses
  - Buses are the paths which digital signals are transferred.
  - Three types of buses: address, data and control.
  - Could be tracks on PCB or wires in a ribbon cable.

Components of a Microprocessor System

- Memory

Memory Type		Description
ROM	Read Only Memory	Non-volatile memory. Can be read but not written to during execution.
PROM	Programmable ROM	Same as ROM but can be programmed ONCE by the user.
EPROM	Erasable and PROM	Can be programmed more than once. Content is erased by using UV light.
EEPROM	Electronically EPROM	Same as EPROM but content is erased using high voltage instead of UV light.
RAM	Random Access Memory	Volatile memory that requires power to hold data.
DRAM	Dynamic RAM	A RAM that uses capacitors. Data must be refreshed due to charge leakage.
SRAM	Static RAM	A RAM that does not need to be refreshed provided power is applied. Faster but more expensive than DRAM.

Microcontroller vs. Microprocessor System

Microprocessor System

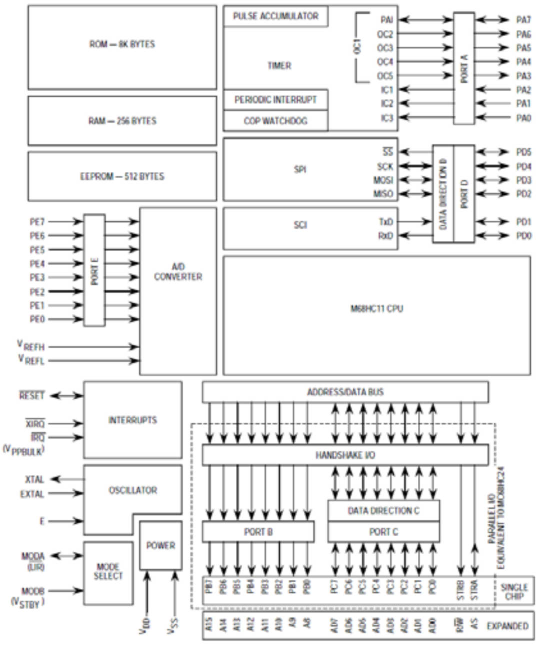
- Memory, I/O components are on separate integrated circuits (IC) or chips.
- Usually have high processing power and memory.
- Example: PC

Microcontroller

- All in one IC.
- Lower processing power and memory.
- Low cost
- Example: appliances, automotive, etc.

M68HC11

- 2 MHz 8-bit microcontroller
- 8 KB ROM
- 512 bytes EEPROM
- 256 bytes RAM
- 8-bit analog-to-digital (A/D) converter



Programming Microcontrollers

- The processor ONLY understands *machine code*.
  - A set of binary instructions.
  - Very difficult to program, difficult to understand, tedious, and prone to errors.
- Low-level language (e.g. *Assembly* language)
  - Simple instruction set (e.g. ADD A, LDA B, etc.) specific to each microcontroller
  - Assembler program converts it to machine code.
- High-level language (e.g. C, C++, FORTRAN, Basic)
  - Easier to understand
  - Program is more portable for use on different microcontrollers
  - Requires compiler to convert to machine code (not always available)