Welcome to CS684: Embedded Systems Labs!

Instructors: Prof. Kavi Arya and Prof. Paritosh Pandya

Lab Schedule:

Sr. No	Lab	Deadline
0	Installation of Software	21-01-2021
1	Bar-graph LEDs and Interrupt Switch Interfacing	18-02-2021
2	Speed Control using Phase Correct PWM Mode	18-02-2021
3	ADC Interfacing	18-02-2021
4	Case Study: Adaptive Cruise Control (Embedded C)	18-02-2021
5	Introduction to RTOS	-
6	Case Study: Adaptive Cruise Control (Statecharts)	
7	Case Study: Adaptive Cruise Control (Lustre)	

Installation Guide

This document contains instructions to install following software/libraries on **Ubuntu OS**:

- **SimulIDE 0.4.14**
- **Eclipse IDE for C/C++ Developers**
- **CoppeliaSim 4.0.0**

> The installation of all software/libraries has been tested on **Ubuntu 16.04** and **18.04**. We recommend you to use one of these versions of **Ubuntu OS**. These software have to be installed **ONLY ON 64-bit OS**.

> Refer the Video for installing Ubuntu on Virtual Machine

After installation, follow the steps in **AVR Building Tool** which will be used to compile and generate hex file from microcontroller code.

SimulIDE:

Description:

SimulIDE is a simple real time electronic circuit simulator, to learn and experiment with simple electronic circuits and microcontrollers, supporting PIC, AVR and Arduino. In this course, we will use it for simulating AVR projects.

Installation Steps:

- Download **SimulIDE Linux Appimage** (*version 0.4.14*) from here

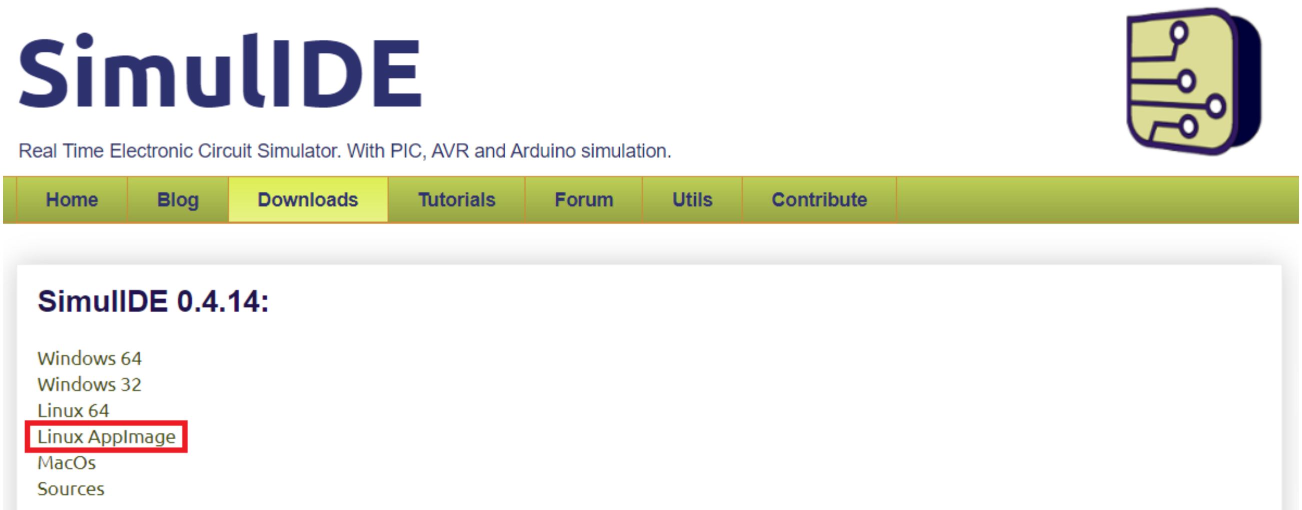


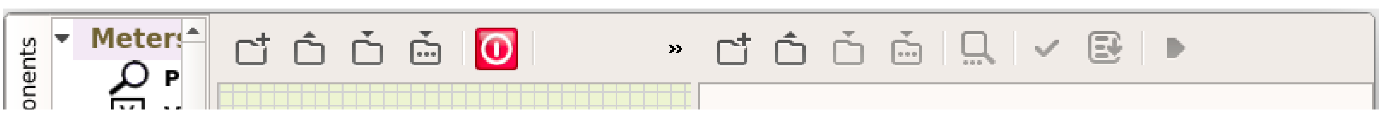
Figure: SimulIDE version selection

- Open Terminal and navigate to the directory where this file was downloaded. Run the following command:

```
chmod +x ./*.AppImage
```

This command will make an Appimage as executable.

- Then double-click the AppImage in the file manager to open SimulIDE.



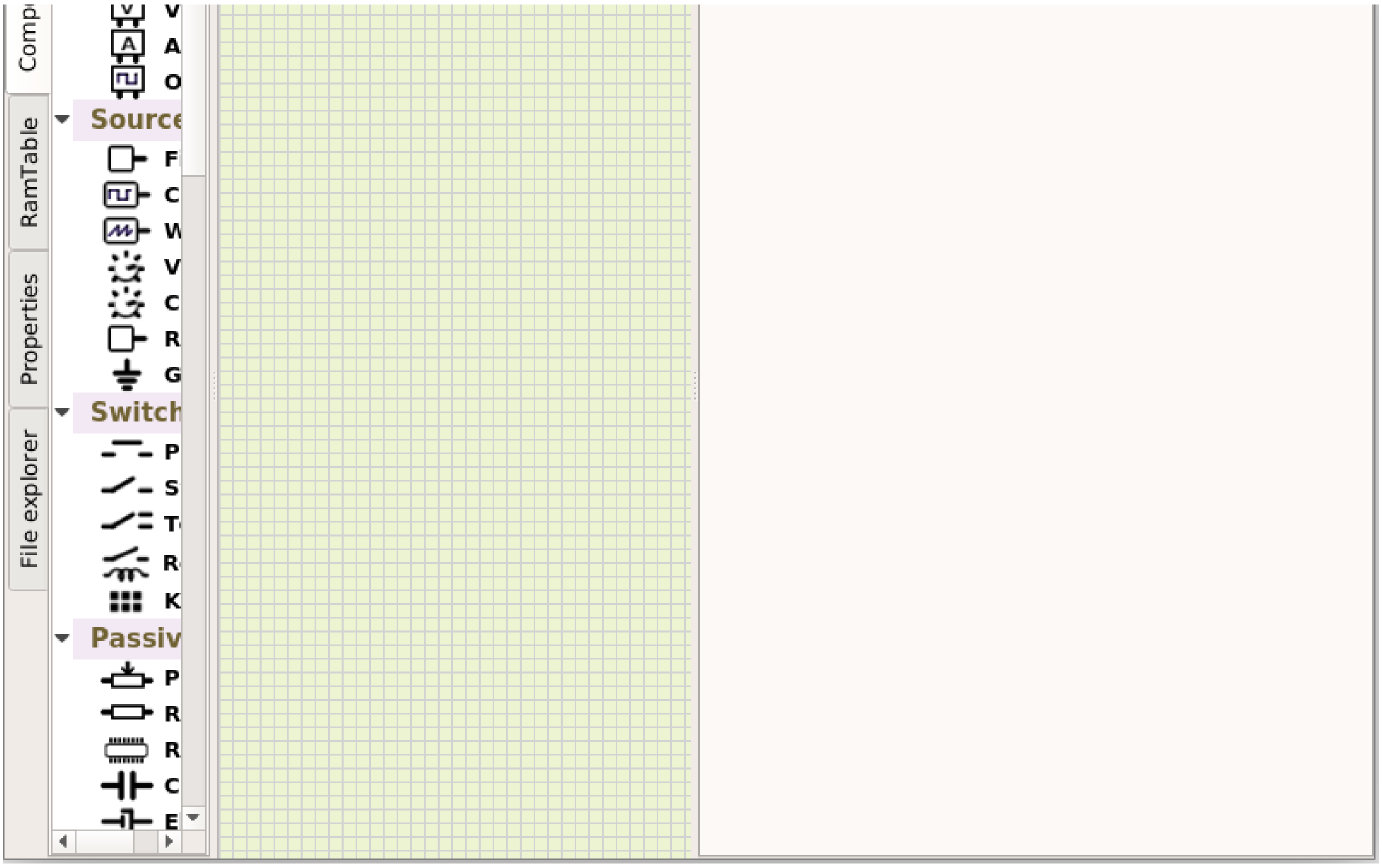


Figure: SimulIDE software launch

Eclipse:

Description: Eclipse is an integrated development environment used in computer programming. It contains a base workspace and an extensible plug-in system for customising the environment. We will be using Eclipse for writing our codes.

Installation Steps:

- Download **Eclipse IDE for C/C++ Developers** for **Ubuntu 64-bit OS** from here (*file size - 358MB*). It will download as **.tar.xz** (compressed zip) file.
- Open Terminal and navigate to the directory where this file was downloaded. Run the following command:

```
tar -xf eclipse-cpp-2020-12-R-linux-gtk-x86_64.tar.gz
```

  This command will decompress and extract the Eclipse software to the folder named **eclipse-cpp-2020-12-R-linux-gtk-x86_64** in the same directory.

- Now type the below commands in sequence to launch CoppeliaSim.

```
cd eclipse
./eclipse
```

- You will see the output as shown in following figures. Eclipse will open with the default scene loaded.

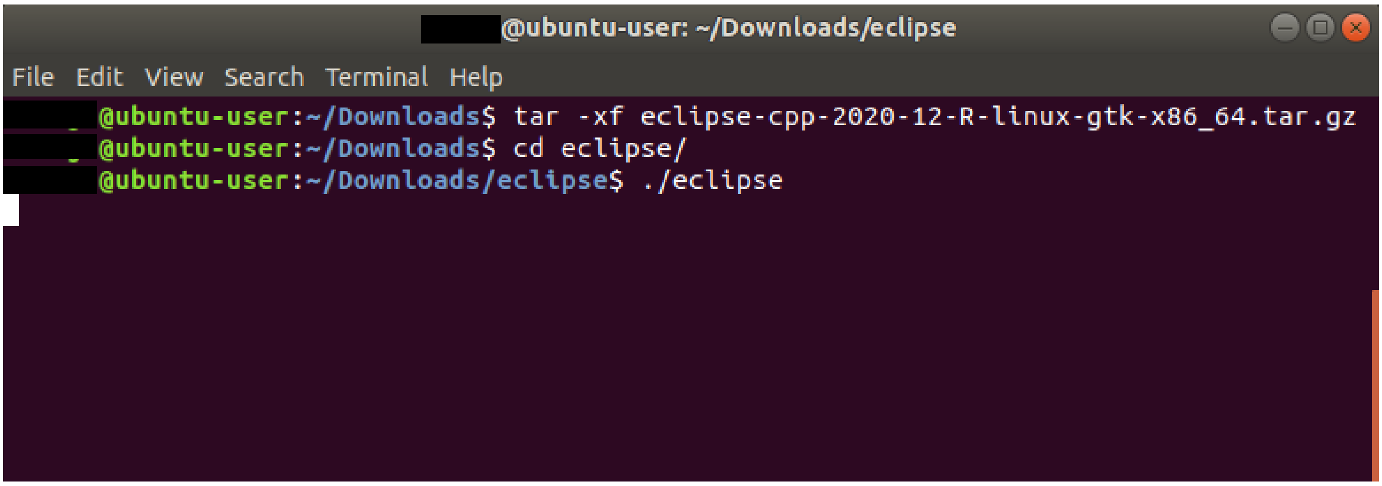


Figure: Extract Eclipse and launch it

Choose the desired location for workspace or continue with the default directory. On launching Eclipse, welcome screen will appear. After closing it you will see the following screen.

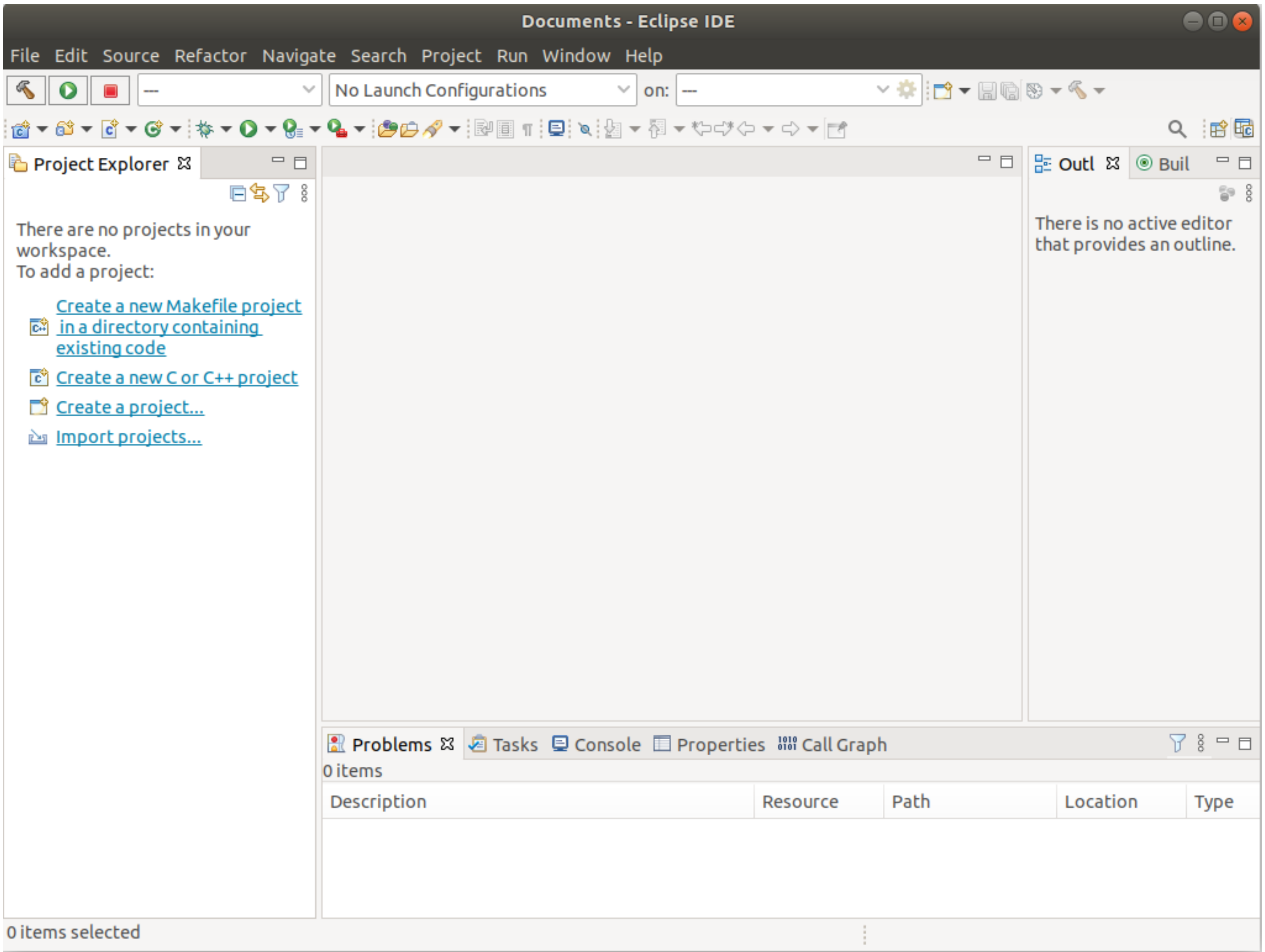


Figure: Eclipse software launch

CoppeliaSim:

Description:

The robot simulator CoppeliaSim is based on a distributed control architecture: each object/model can be individually controlled via an embedded script, a plugin, a ROS or BlueZero node, a remote API client, or a custom solution. This makes CoppeliaSim very versatile and ideal for multi-robot applications. Controllers can be written in C/C++, Python, Java, Lua, Matlab or Octave.

Installation Steps:

- Download **CoppeliaSim Edu 4.0.0** for **Ubuntu 18.04** (*64-bit OS*) from here (*file size - 152MB*). It will download as **.tar.xz** (compressed zip) file.

> **Note:** To download CoppeliaSim for **Ubuntu 16.04** (*64-bit OS*), click here (*file size - 144MB*).

- Open Terminal and navigate to the directory where this file was downloaded. Run the following command:

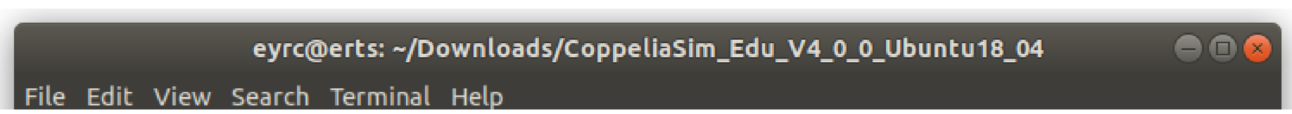
  ```
  tar -xf CoppeliaSim_Edu_V4_0_0_Ubuntu18_04.tar.xz
  ```

  This command will decompress and extract the CoppeliaSim software to the folder named **CoppeliaSim_Edu_V4_0_0_Ubuntu18_04** in the same directory.

- Now type the below commands in sequence to launch CoppeliaSim.

  ```
  cd CoppeliaSim_Edu_V4_0_0_Ubuntu18_04
  ./coppeliaSim.sh
  ```

- You will see the output as shown in following figures. CoppeliaSim will open with the default scene loaded.



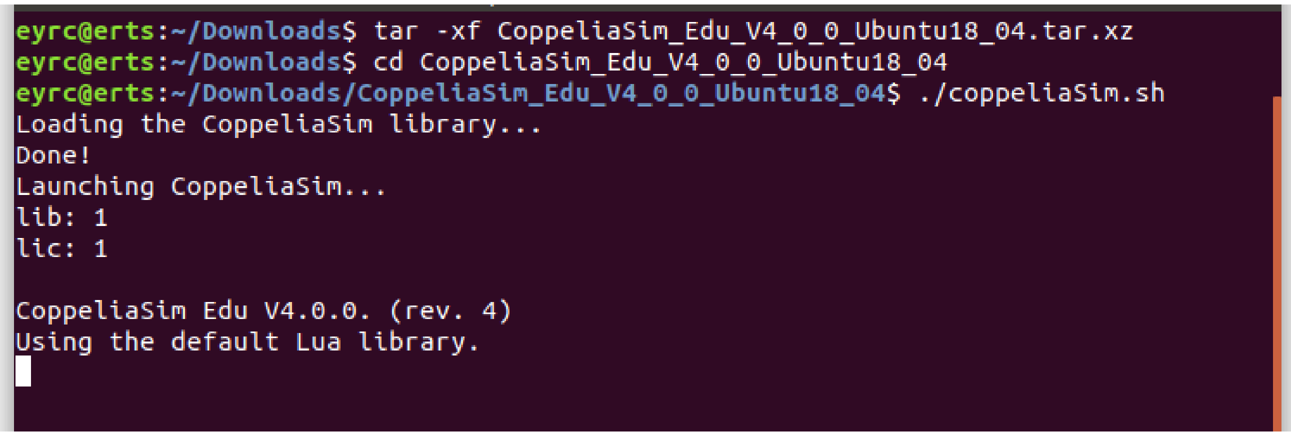


Figure: Extract CoppeliaSim and launch it

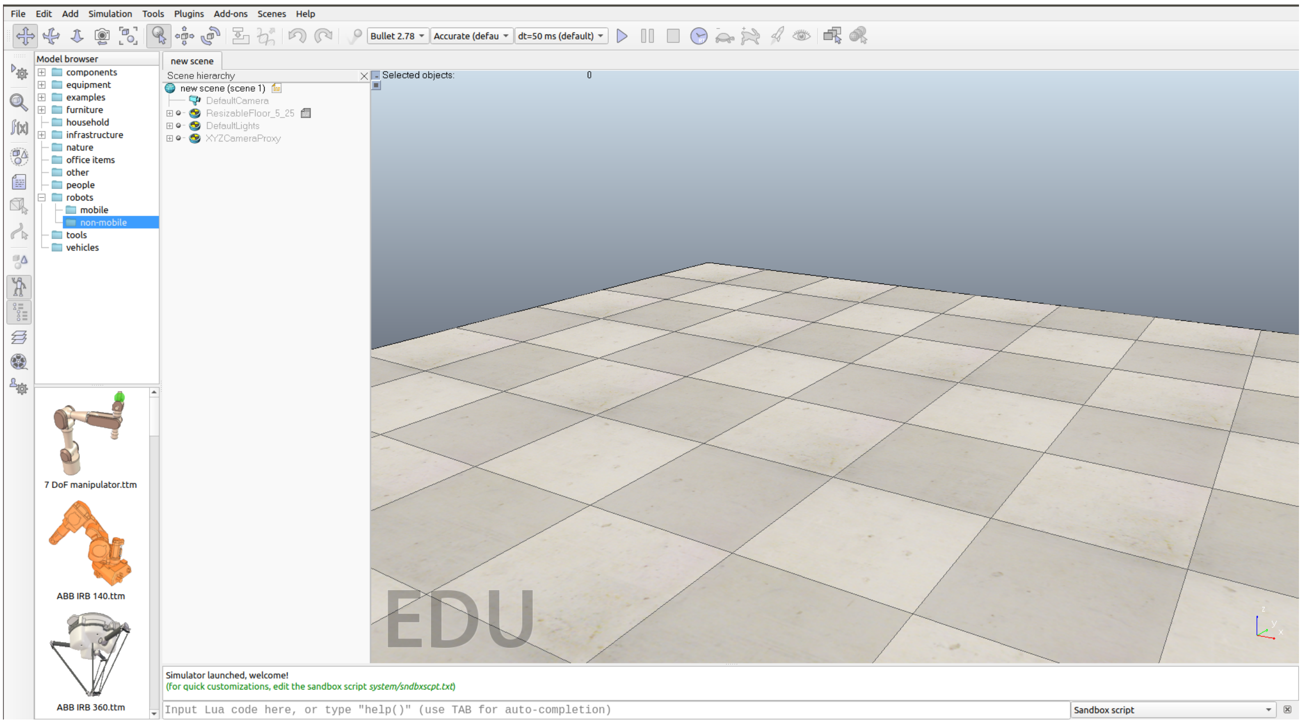


Figure: CoppeliaSim software launch

AVR Building Tool:

Description:

This is user generated executable file. The file will be used to compile and build AVR codes and generate hex file.

Download **avr_hex_file_generator** executable file. Right-click on the hyperlinks and select **Save Link As...** option to download.

Set a $PATH of executable file in environment variable.

Follow the steps to set environment variable:

- Open Terminal and navigate to home directory:

```
cd $HOME
```

- Run the following command:

```
sudo apt-get install gcc-avr binutils-avr gdb-avr avr-libc avrdude
```

- Open the .bashrc file in any editor:

```
sudo nano .bashrc
```

- Add the following line to the file at the end.

```
export PATH="$HOME/<Path_to_executable_file>:$PATH"
```

- Save the file and exit. Use the source command to force Linux to reload the .bashrc file which normally is read only when you log in each time.

```
source .bashrc
```

- Check that path is correctly added in the environment variable:

```
echo $PATH
```

CS684: Embedded System Course

Resources

This document consists of links of video tutorials needed to complete the labs. It also consists of few additional modules for deeper understanding.

> **Note:** All the resources provided below are based on Firebird V robot which consists of ATmega2560 microcontroller. But for the lab sessions, we will be working on ATmega328 controller (Arduino Uno) given that concepts remain same across both the controllers (belong to same family).

> **Reference Code Examples:** Experiments of Firebird V

Additional Module: Introduction to Firebird V Robot

Lab 1: Bar-graph LEDs and Interrupt Switch Interfacing

Module 1: I/O Interfacing

- Understand the function of I/O ports and the associated registers.
- Interface I/O peripherals like Switch, Buzzer and Bar graph LEDs.

Module 2: Masking

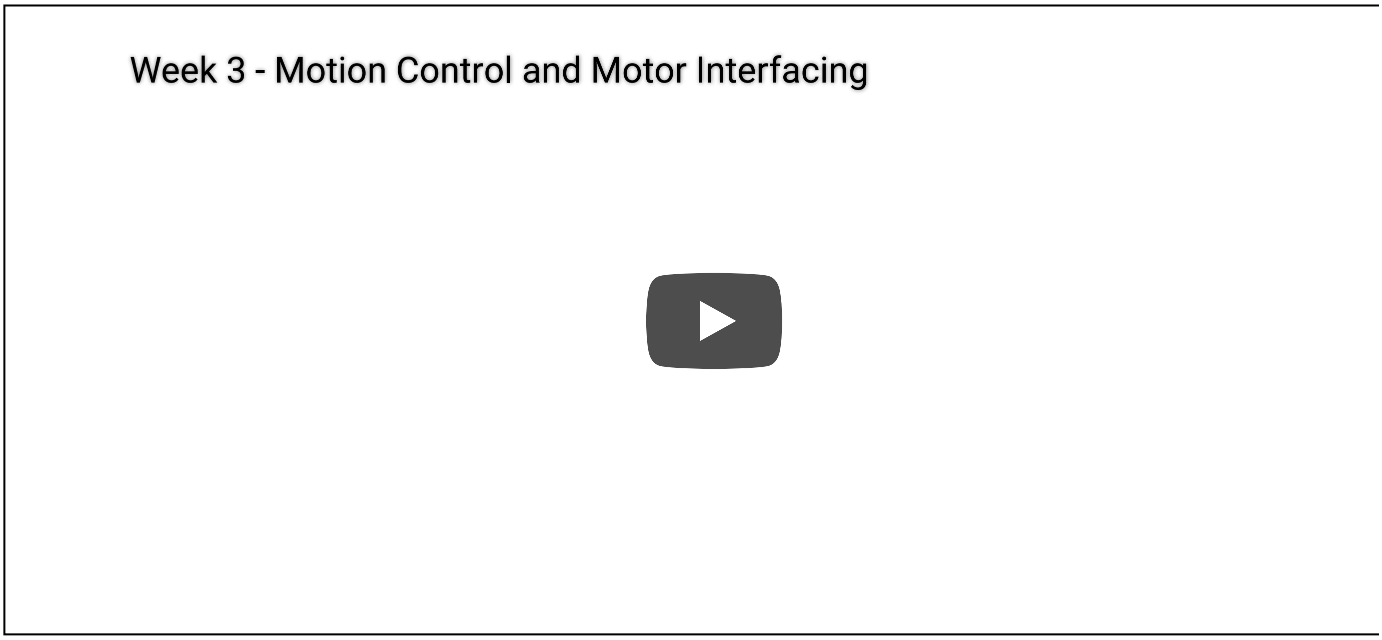
- Understand the concept of Masking with the help of I/O interfacing.
- Need of using AND and OR operators with the help of few examples.

Additional Module: LCD Interfacing

Lab 2: Speed Control using 8 bit Phase Correct PWM Mode

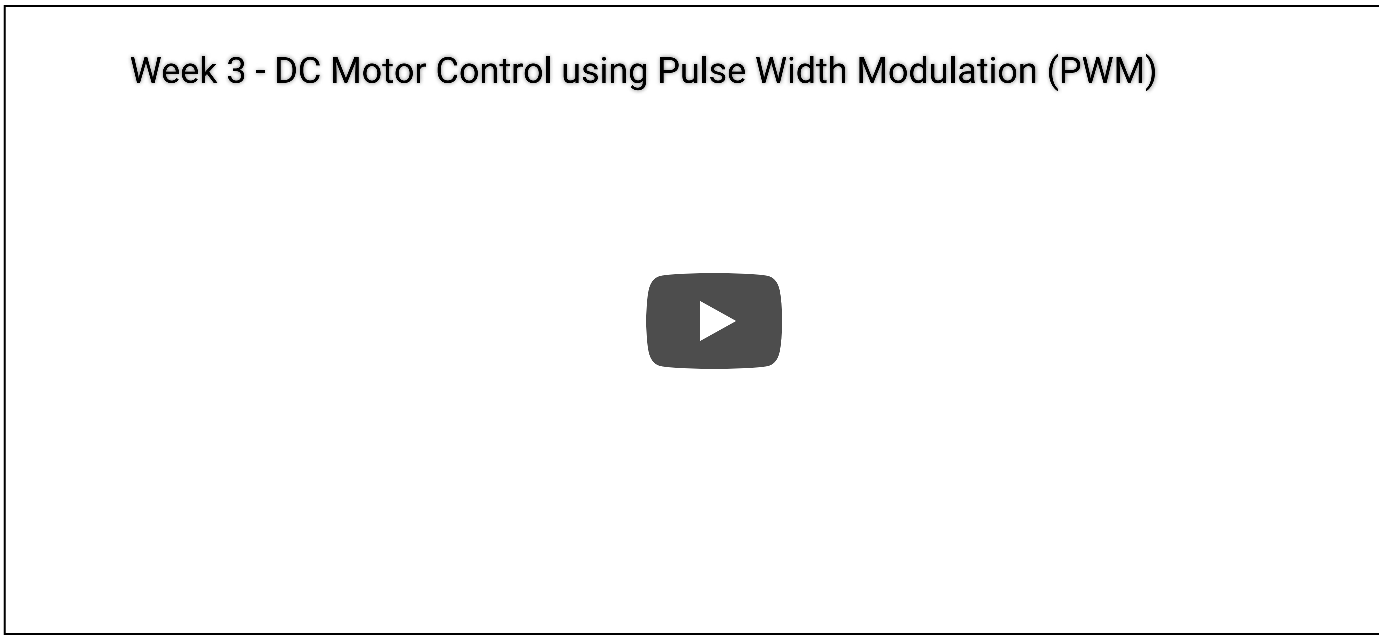
Module 3: Motor Interfacing

- Direction control of DC motors present on Firebird V Robot.
- Understanding the use of L293D motor driver IC.



Module 4: Pulse Width Modulation

- Understanding Timers and associated registers.
- Speed control of the robot and brightness control of the LED.



Additional Modules:

- Interrupts
- Position Encoder Interrupt

Lab 3: ADC Interfacing

Module 5: ADC Interfacing

- Interface white line sensors and Proximity sensor.
- Understand ADC (Analog to Digital conversion) on controller.

Additional Module: Serial Communication

Lab 5: Real-Time Operating Systems (RTOS)

Following is the link to download the **zip file** of examples on concepts of RTOS implemented using **FreeRTOS API** built with **ESP32** module.

RTOS Examples on ESP32

Project: Search and Rescue

Task 1 PPT - Presented in class of ***22nd March, 2021***

Firebird V related Resources:

- Firebird V Hardware and Software Manuals
- ATmega2560 Datasheet
- Experiments of Firebird V
- Color Sensor Interfacing on Firebird V
- Serial Communication between Firebird V and ESP 32

Simulation related Resources

Simulation related Resources

- CoppeliaSim Scene File: search_n_rescue.ttt
- Eclipse Project Folder for Task 1

IoT Related Resources

- Mocking RPC

CS684: Embedded System Course

Lab 1: Bar-graph LEDs and Interrupt Switch Interfacing

LEARNING RESOURCES

This lab requires understanding of I/O interfacing with AVR microcontroller and basic knowledge about Masking. Refer the **Resources** file.

AIM

In this lab, you will interface the Bar-graph LEDs and the Interrupt Switch to ATmega2560. The aim of this lab is to get you familiar with the configuring and interfacing of Input and Output devices.

Your task is to toggle the status of 2 Bar-graph LEDs depending on whether the Interrupt Switch is pressed or released.

The program is provided as **Eclipse** project. But, the program contains few incomplete functions which you would have to complete as per the instructions present in the comments.

CONNECTIONS

- Interrupt Switch : **PD2**
- Bar-graph LED:
  - LED 2 --> **PB1**
  - LED 6 --> **PB5**
  - LED 8 --> **PB0**

PROCEDURE

Step-1: Download **lab1_sw_led** zip folder. Right-click on the hyperlinks and select **Save Link As...** option to download. Extract the zip file. Start Eclipse, click on *File* > *Open Projects From File System* > *Directory* and browse for **lab1_sw_led** folder to open the project.

Step-2: In **src/eBot_Sandbox.cpp**, complete the **toggle_leds_on_sw_press** function to achieve the following:

1. Check whether the Interrupt Switch is pressed or not.
2. If the Interrupt Switch is pressed, only the **2nd** LED will turn OFF and the **8th** LED will turn ON.
3. If the Interrupt Switch is not pressed, the **2nd** LED will remain ON and only **8th** LED will turn OFF.

> **Note:** While completing, you have to use following functions defined in **eBot_MCU_Predef.h** header file:

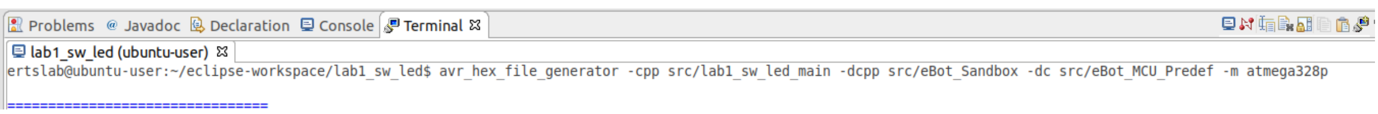
1. bool interrupt_switch_pressed (void);
2. void turn_on_bar_graph_led (unsigned char);
3. void turn_off_bar_graph_led (unsigned char);

Step-3: You will notice some pre-written function stubs in **src/eBot_MCU_Predef.c** included for your assistance related to Bar-graph LEDs and Interrupt Switch. Complete the pre-written functions with the help of comments provided.

Step-4: In the Package Explorer (left pane), right click on **lab1_sw_led** folder and select ***Show in Local Terminal --> Terminal***(bottom pane). Type the following command in the terminal:

```
avr_hex_file_generator -cpp src/lab1_sw_led_main -dcpp src/eBot_Sandbox -dc
src/eBot_MCU_Predef -m atmega328p
```

If there are no errors present in your program, the project will get compiled correctly and you will get the message as shown below, Also, **build** folder will be generated in project directory that will contain **lab1_sw_led_main.hex** file along with other build files.



```
 ### AVR Hex File Generator ###
=================================

Compiling the main C++ file:
Main C++ file compiled successfully without any errors and 'build/lab1_sw_led_main.o' is generated correctly.

Compiling all the provided dependent C files:
Dependent C file compiled successfully without any errors and 'build/eBot_MCU_Predef.o' is generated correctly.

Compiling all the provided dependent C++ files:
Dependent C++ file compiled successfully without any errors and 'build/eBot_Sandbox.o' is generated correctly.

Linking the main C++ file and dependent files (if any):
Main C++ file linked successfully without any errors and 'build/lab1_sw_led_main.elf' is generated correctly.

Generating the Hex file:
Hex file 'build/lab1_sw_led_main.hex' generated successfully without any errors.

ertslab@ubuntu-user:~/eclipse-workspace/lab1_sw_led$
```

Step-5: Load the hex file on the **SimulIDE circuit (AVR_simulator.simu)** once your program gets build successfully. Save the project. Ensure that code performs as expected. **NOTE:** Getting **Build succeeded** output doesn't mean that your program will give the expected output as it can contain logical errrors.

MACROS to use

- In the skeleton code **eBot_MCU_Predef.c** we have included a header file named, **eBot_MCU_Predef.h** which consists of the declaration of the following constants (*for atmega328p controller*):

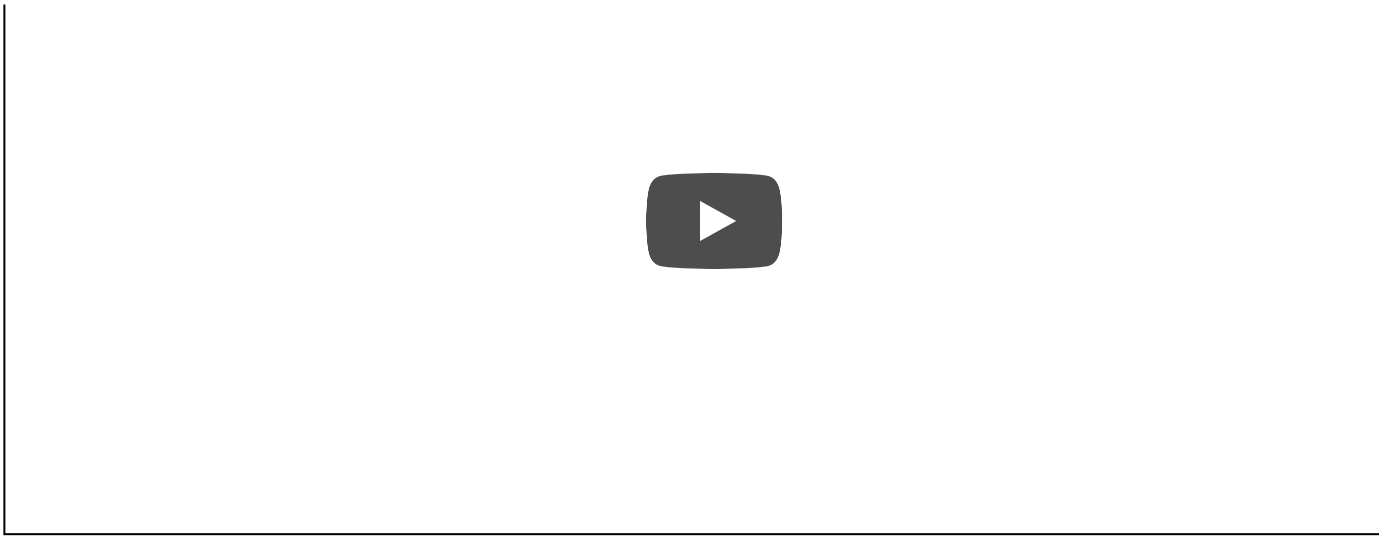
```
//For Interrupt Switch:

#define         interrupt_sw_ddr_reg            DDRD
#define         interrupt_sw_port_reg           PORTD
#define         interrupt_sw_pin                2                // PD2

//For Bar-graph LED:

 #define        bar_graph_led_ddr_reg           DDRB
 #define        bar_graph_led_port_reg  PORTB
 #define        bar_graph_led_2_pin             1                // PB1
 #define        bar_graph_led_8_pin             0                // PB0
 #define        bar_graph_led_6_pin             5                // PB5
```

- You have to use these constants declared in **eBot_MCU_Predef.h** and the Masking Operators to complete the lab.

Simulation: EXPECTED OUTPUT

Software required: SimulIDE AppImage (refer Installation_Instructions provided to you)

Switch LED Toggle Output Video



CS684: Embedded System Course

Lab 2: Speed Control using 8 bit Phase Correct PWM Mode

LEARNING RESOURCES

This lab requires understanding of motor interfacing with AVR microcontroller and basic knowledge about Pulse Width Modulation. Refer the **Resources** file.

AIM

In this lab, you will increase and decrease the speed of motors using Phase Correct PWM Mode. The aim of this lab is to get you familiar with one of the PWM mode available on the ATmega2560.

In this lab, you will interface the L293D motor driver and the Timer connected to the ATmega2560.

The program is provided as **Eclipse** project. But, the program contains few incomplete functions which you would have to complete as per the instructions present in the comments.

CONNECTIONS

- Motors are connected to the Microcontroller through L293D Motor Driver IC.

  Motors Pin Microcontroller Pin

  - RB -> **PC3**

- RB --> **PC3**
- RF --> **PC2**
- LF --> **PC1**
- LB --> **PC0**

- PWM Pins of the Microcontroller are connected to the L293D Motor Driver IC.

  PWM Pin Microcontroller Pin

  - Left Motor --> **PD5**
  - Right Motor --> **PD6**

PROCEDURE

Step-1: Download **lab2_motor_pwm** zip file. Right-click on the hyperlinks and select **Save Link As...** option to download. Extract the zip file. Start Eclipse, click on *File --> Open Projects From File System --> Directory* and browse for **lab2_motor_pwm** folder to open the project.

Step-2: In **src/eBot_Sandbox.cpp**, complete the **traverse_s_shape** function to achieve the following:

1. Move the robot in forward direction.
2. In the while loop, set the speed as complement of each other for the left and right motors and increment it by one every ten milliseconds till 255.

> **Note:** While completing, you have to use following functions defined in **eBot_MCU_Predef.h** header file:

1. void forward (void);
2. void velocity (unsigned char, unsigned char);

Step-3: Build the project (click on *Project --> Build Project*). If there are no errors present in your program, the project will get build successfully and you will get the message (*on console at bottom pane*) as shown below,

```
**========= Build Finished. 0 errors, 0 warnings  =========**
```

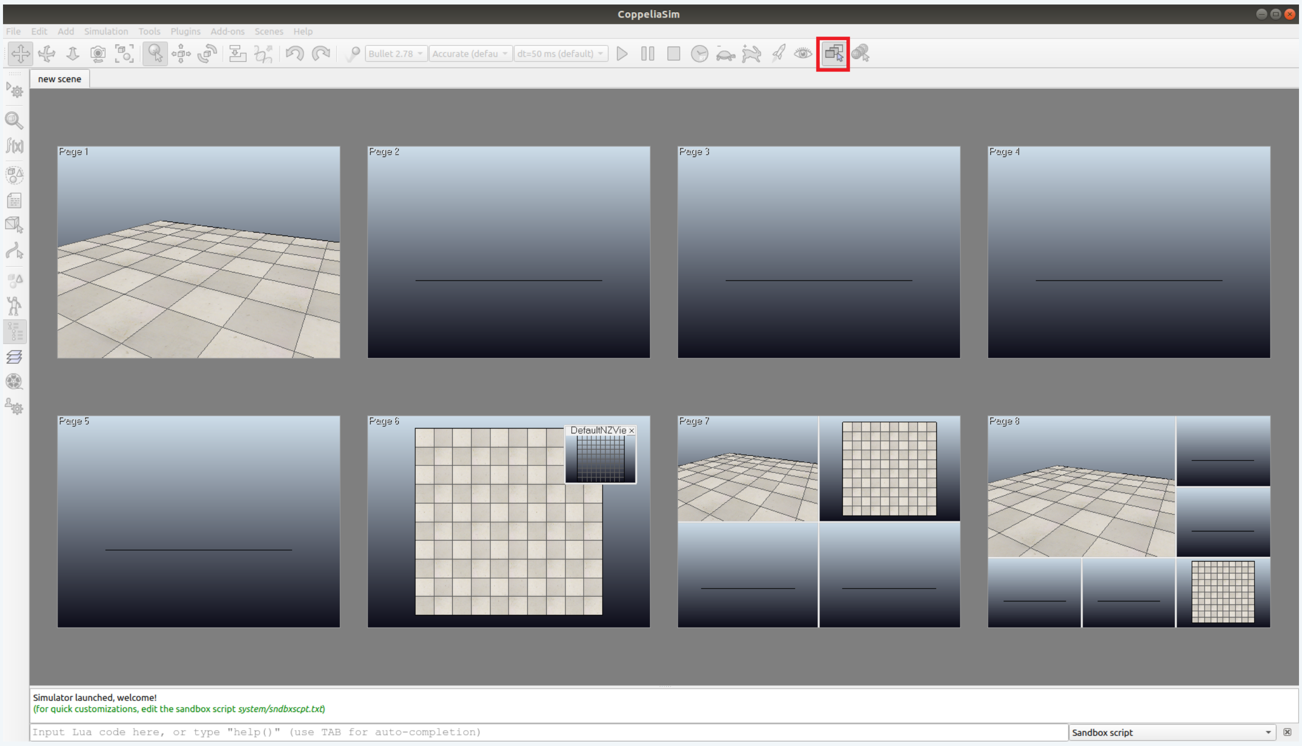
Step-4: Check the behaviour of the robot on CoppeliaSim.

- Download the **eBot.ttm** model. Start CoppeliaSim, click on *File --> Load Model* and browse for **eBot.ttm** file. This will load the robot model on the scene. (shown in *Expected Output - CoppeliaSim* video).

- In Eclipse, right click on **lab2_motor_pwm** folder (Project Explorer - left pane) and select *Run As --> Local C/C++ Application*. If no errors are encountered, following message will be displayed onto the console (bottom pane):

```
Connection Success ...
0
0
0

 Please Enter Y to Start Simulation:      Y
```

Enter 'Y' on console. This will start the simulation in CoppeliaSim. You can verify the output by referring to *Expected Output - CoppeliaSim* video.

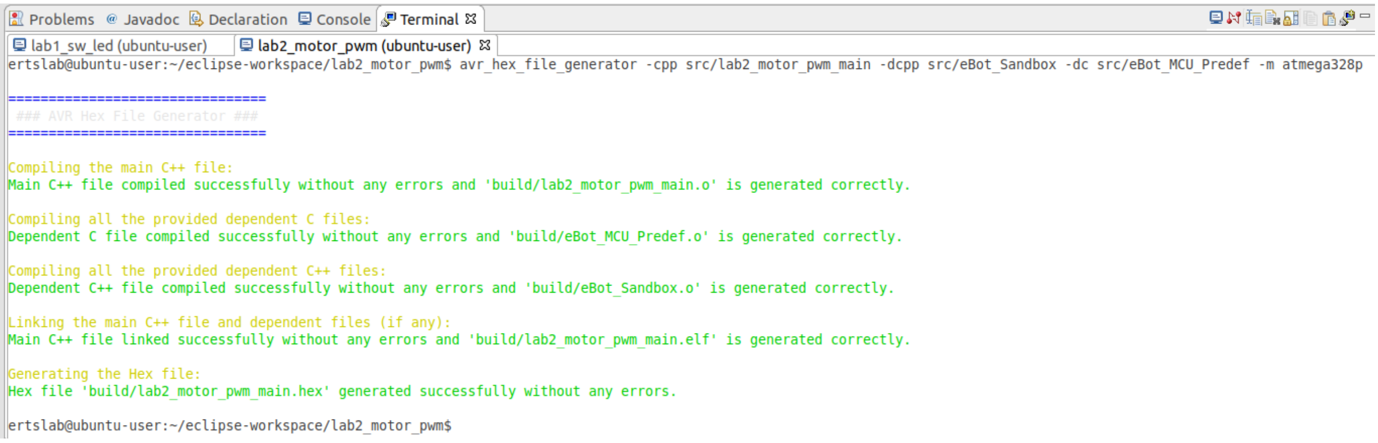
> **Note:** For better angle, you can change the perspective of the scene using page selector.
>
> 
>

Step-5: You will notice some pre-written function stubs in **src/eBot_MCU_Predef.c** included for your assistance related to motor and timer. Complete the pre-written functions with the help of comments provided.

Step-6: In the Package Explorer (left pane), right click on **lab2_motor_pwm** folder and select *Show in Local Terminal* --> *Terminal* (bottom pane). Type the following command in the terminal:

```
avr_hex_file_generator -cpp src/lab2_motor_pwm_main -dcpp src/eBot_Sandbox -dc
src/eBot_MCU_Predef -m atmega328p
```

If there are no errors present in your program, the project will get compiled correctly and you will get the message as shown below, Also, **build** folder will be generated in project directory that will contain **lab2_motor_pwm_main.hex** file along with other build files.



Step-7: Load the hex file on the **SimulIDE circuit (AVR_simulator.simu)** once your program gets build successfully. Save the project. Ensure that code performs as expected. **NOTE:** Getting **Build succeeded** output doesn't mean that your program will give the expected output as it can contain logical errrors.

MACROS to use

- In the skeleton code **eBot_MCU_Predef.c** we have included a header file named, **eBot_MCU_Predef.h** which consists of the declaration of the following constants (*for atmega328p controller*):

```
// Motor direction registers and pins
```

```
    #define             motors_dir_ddr_reg              DDRC
    #define             motors_dir_port_reg             PORTC
    #define      motors_RB_pin                  PC3             // 3
    #define      motors_RF_pin                  PC2             // 2
    #define      motors_LF_pin                  PC1             // 1
    #define      motors_LB_pin                  PC0             // 0

    // Motor enable registers and pins
    #define             motors_pwm_ddr_reg              DDRD
    #define             motors_pwm_port_reg             PORTD
    #define             motors_pwm_R_pin                PD6             // 6
    #define             motors_pwm_L_pin                PD5             // 5

    // Timer / Counter registers
    #define             TCNTH_reg                               TCNT0H  // Timer
/ Counter 0 High Byte register
    #define             TCNTL_reg                               TCNT0L  // Timer
/ Counter 0 Low Byte register
    #define             OCRAL_reg                               OCR0AL  //
Output Compare Register 0A Low Byte
    #define             OCRAH_reg                               OCR0AH  //
Output Compare Register 0A High Byte
    #define             OCRBL_reg                               OCR0BL  //
Output Compare Register 0B Low Byte
    #define             OCRBH_reg                               OCR0BH  //
Output Compare Register 0B High Byte
    #define             TCCRA_reg                               TCCR0A  // Timer
/ Counter Control Register 0A
    #define             TCCRB_reg                               TCCR0B  // Timer
/ Counter Control Register 0B

    // Bits of compare output mode in the TCCRnA register ( Timer / Counter 'n'
Control Register A, where n = 0, 1, 2, 3, 4, 5 )
    #define             COMA1_bit                       COM0A1  // 7 (Compare
Output Mode bit 1 for Channel A)
    #define             COMA0_bit                       COM0A0  // 6 (Compare
Output Mode bit 0 for Channel A)
    #define             COMB1_bit                       COM0B1  // 5 (Compare
Output Mode bit 1 for Channel B)
    #define             COMB0_bit                       COM0B0  // 4 (Compare
Output Mode bit 0 for Channel B)

    // Bits of waveform generation mode in the TCCRnA and TCCRnB register ( Timer
/ Counter 'n' Control Register A/B, where n = 0, 1, 2, 3, 4, 5 )
    #define             WGM0_bit                                WGM00   // 0
(Waveform Generation Mode bit 0)
    #define             WGM1_bit                                WGM01   // 1
(Waveform Generation Mode bit 1)
    #define             WGM2_bit                                WGM02   // 2
(Waveform Generation Mode bit 2)
    #define             WGM3_bit                                WGM03   // 3
(Waveform Generation Mode bit 3)

    // Bits of clock select mode in the TCCRnB register ( Timer / Counter 'n'
Control Register B, where n = 0, 1, 2, 3, 4, 5 )

    #define             CS0_bit                                 CS00    // 0
(Clock Select bit 0)
    #define             CS1_bit                                 CS01    // 1
```

```
(Clock Select bit 1)
   #define          CS2_bit                    CS02    // 2
(Clock Select bit 2)
```

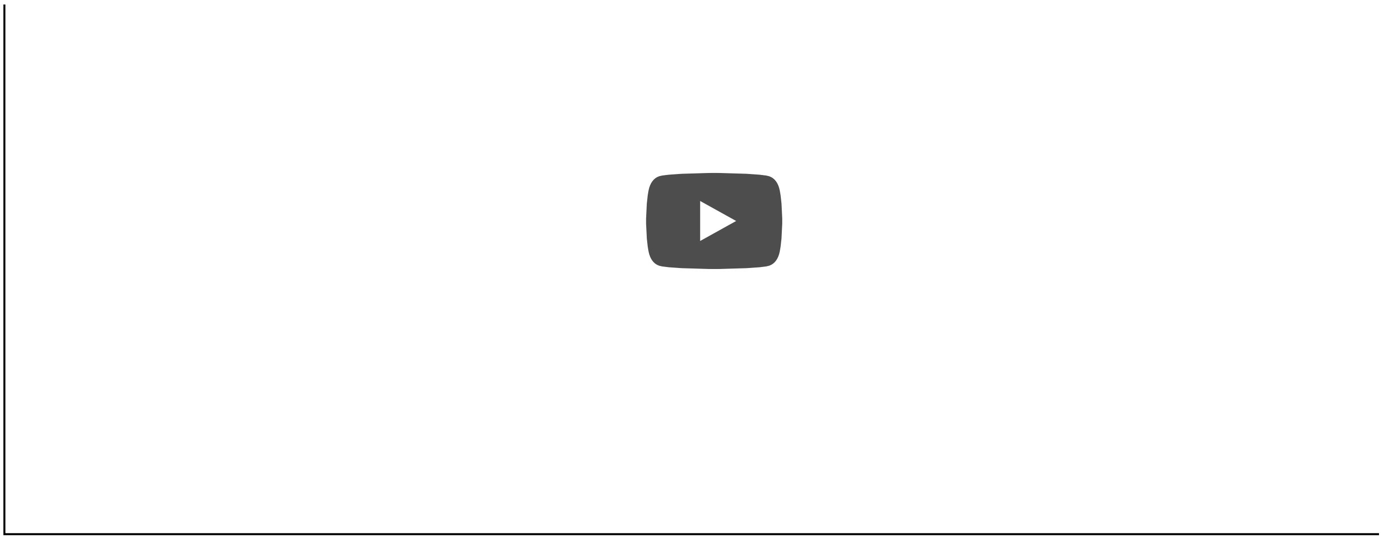
- You have to use these constants declared in **eBot_MCU_Predef.h** and the Masking Operators to complete the lab.

EXPECTED OUTPUT - CoppeliaSim

Software required: CoppeliaSim and Eclipse (refer Installation_Instructions provided to you)

EXPECTED OUTPUT - SimulIDE

Software required: SimulIDE AppImage (refer Installation_Instructions provided to you)



CS684: Embedded System Course

Lab 3: ADC Interfacing

LEARNING RESOURCES

This lab requires understanding of sensors interfacing with AVR microcontroller and basic knowledge about Analog to Digital Conversion. Refer the **Resources** file.

AIM

In this lab, you will interface ADC with the micro-controller. The aim of this lab is to get you familiar with ADC present on ATmega2560.

In this lab, you will interface the White Line and IR Proximity sensors. Your task is to get the **8-bit** ADC result from the **three white line** sensors and **5th IR proximity** sensor in Single Conversion Mode and display the ADC converted digital values on UART Serial Terminal.

The program is provided as **Eclipse** project. But, the program contains few incomplete functions which you would have to complete as per the instructions present in the comments.

CONNECTIONS

- Left White Line Sensor : **PC1 [ADC Channel 1]**
- Center White Line Sensor : **PC0 [ADC Channel 0]**

- Right White Line Sensor : **PC2 [ADC Channel 2]**
- 5th IR Proximity Sensor : **PC5 [ADC Channel 5]**

PROCEDURE

Step-1: Download **lab3_adc** zip file. Right-click on the hyperlinks and select **Save Link As...** option to download. Extract the zip file. Start Eclipse, click on *File* --> *Open Projects From File System* --> *Directory* and browse for **lab3_adc** folder to open the project.

Step-2: In **src/eBot_Sandbox.cpp**, complete the **send_sensor_data** function to achieve the following:

Read data from 3 white line sensors and IR proximity sensor. Print IR proximity sesnor data on the terminal.

> **Note:** While completing, you have to use following functions defined in **eBot_MCU_Predef.h** header file:

1. unsigned char convert_analog_channel_data(unsigned char sensor_channel_number);
2. int print_ir_prox_5_data(unsigned char);

Step-3: Build the project (click on *Project* --> *Build Project*). If there are no errors present in your program, the project will get build successfully and you will get the message (*on console at bottom pane*) as shown below,

```
**========= Build Finished. 0 errors, 0 warnings  =========**
```

Step-4: Check the behaviour of the robot on CoppeliaSim.

- Download the **lab3_adc.ttt** scene. Start CoppeliaSim, click on *File* --> *Open Scene* and browse for **lab3_adc.ttt** file to open the scene (shown in *Expected Output - CoppeliaSim* video).
- In Eclipse, right click on **lab3_adc** folder (Project Explorer - left pane) and select *Run As --> Local C/C++ Application*. If no errors are encountered, following message will be displayed onto the console (bottom pane):

```
Connection Success ...
0
0
0

 Please Enter Y to Start Simulation:      Y
```

Enter 'Y' on console. This will start the simulation in CoppeliaSim and following message will be displayed on the console:

```
Simulation started correctly.
Initialized all sensors in the current CoppeliaSim scene.
```

```
Enter 'Y' / 'y' to take the next sensor reading or 'Q' / 'q' to quit: y
```

You can check the reading of IR proximity sensor by entering 'Y' / 'y'. Change the position of the robot (in CoppeliaSim) and observe the readings (in Eclipse).

> **Note:** To change the position of the robot (in CoppeliaSim) click on *object/item shift* and enter X and Y co-ordinates in position tab.
> To change the orientation of the robot, click on *object/item rotate* and enter alpha, beta and gamma values in orientation tab. (Refer EXPECTED OUTPUT - CoppeliaSim video)

- Verify the readings for following position and orientation:

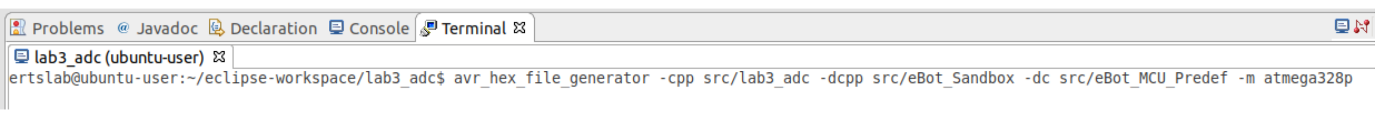
```
pos(X,Y) - (0.2020, 0.2000),   ori(Alpha, Beta, Gamma) - (0, 0, 90)  -->
reading(127)
pos(X,Y) - (0.2020, -0.2000),  ori(Alpha, Beta, Gamma) - (0, 0, -90) -->
reading(127)
pos(X,Y) - (0.6060, -0.2000),  ori(Alpha, Beta, Gamma) - (0, 0, 0)   -->
reading(70)
pos(X,Y) - (-0.6060, -0.2000), ori(Alpha, Beta, Gamma) - (0, 0, 90)  -->
reading(73)
```

Step-5: You will notice some pre-written function stubs in **src/eBot_MCU_Predef.c** included for your assistance related to motor and timer. Complete the pre-written functions with the help of comments provided.

Step-6: In the Package Explorer (left pane), right click on **lab3_adc** folder and select *Show in Local Terminal* --> *Terminal* (bottom pane). Type the following command in the terminal:

```
avr_hex_file_generator -cpp src/lab3_adc_main -dcpp src/eBot_Sandbox -dc
src/eBot_MCU_Predef -m atmega328p
```

If there are no errors present in your program, the project will get compiled correctly and you will get the message as shown below, Also, **build** folder will be generated in project directory that will contain **lab3_adc_main.hex** file along with other build files.



```
================================
 ### AVR Hex File Generator ###
================================

Compiling the main C++ file:
Main C++ file compiled successfully without any errors and 'build/lab3_adc.o' is generated correctly.

Compiling all the provided dependent C files:
Dependent C file compiled successfully without any errors and 'build/eBot_MCU_Predef.o' is generated correctly.

Compiling all the provided dependent C++ files:
Dependent C++ file compiled successfully without any errors and 'build/eBot_Sandbox.o' is generated correctly.

Linking the main C++ file and dependent files (if any):
Main C++ file linked successfully without any errors and 'build/lab3_adc.elf' is generated correctly.

Generating the Hex file:
Hex file 'build/lab3_adc.hex' generated successfully without any errors.

ertslab@ubuntu-user:~/eclipse-workspace/lab3_adc$
```

Step-7: Load the hex file on the **SimulIDE circuit (AVR_simulator.simu)** once your program gets build successfully. Save the project. Ensure that code performs as expected. **NOTE:** Getting **Build succeeded** output doesn't mean that your program will give the expected output as it can contain logical errrors.

MACROS to use

- In the skeleton code **eBot_MCU_Predef.c** we have included a header file named, **eBot_MCU_Predef.h** which consists of the declaration of the following constants (*for atmega328p controller*):

```
    // For 3 White Line Sensors
    #define      wl_sensors_ddr_reg                          DDRC
    #define      wl_sensors_port_reg                       PORTC
    #define      left_wl_sensor_pin                        1        // PC1
    #define      left_wl_sensor_channel            1       // ADC1 - ADC Channel 1
    #define      center_wl_sensor_pin                      0        // PC0
    #define      center_wl_sensor_channel                  0        // ADC0 - ADC
Channel 0
    #define      right_wl_sensor_pin                       2        // PC2
    #define      right_wl_sensor_channel                   2        // ADC2 - ADC
Channel 2

    // For 5th IR proximity Sensor
    #define      ir_prox_5_sensor_ddr_reg                  DDRC
    #define      ir_prox_5_sensor_port_reg                 PORTC
    #define      ir_prox_5_sensor_pin                      5        // PC5
    #define      ir_prox_5_sensor_channel                  5        // ADC5 - ADC
Channel 5
```

- You have to use these constants declared in **eBot_MCU_Predef.h** and the Masking Operators to complete the lab.

EXPECTED OUTPUT - CoppeliaSim

Software required: CoppeliaSim and Eclipse (refer Installation_Instructions provided to you)

EXPECTED OUTPUT - SimulIDE

Software required: SimulIDE AppImage (refer Installation_Instructions provided to you)

CS684: Embedded System Course

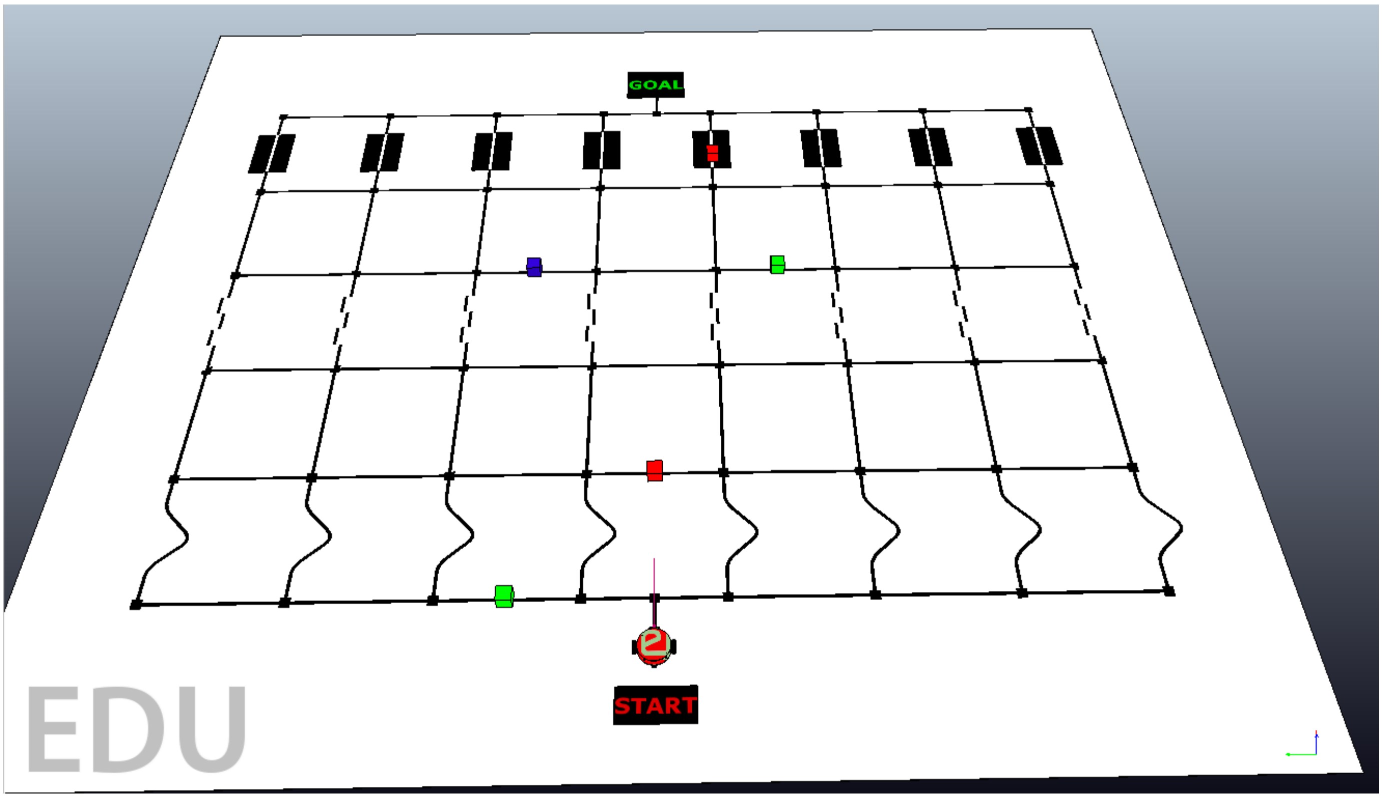
Lab 4: Case Study - Adaptive Cruise Control (Embedded C)

AIM

In this task, you will learn the concept of line following algorithm.

- The arena provided to you is made up of curved, zig-zag, and straight paths with a

- The arena provided to you is made up of curved, zig-zag, and straight paths with a START and GOAL position for the robot as shown in following figure.
- Arena also consists of obstacles randomly placed.
- You have to program the robot which will begin from START position, follow the path avoiding obstacles and stop at the GOAL position.



The program is provided as **Eclipse** project. But, the program contains few incomplete functions which you would have to complete to achive the aim.

PROCEDURE

Step-1: Download **lab4_line_follower_robot** zip folder. Right-click on the hyperlinks and select **Save Link As...** option to download. Extract the zip file. Start Eclipse, click on *File* > *Open Projects From File System* > *Directory* and browse for **lab4_line_follower_robot** folder to open the project.

Step-2: In **src/eBot_Sandbox.cpp**, complete the **traverse_line_to_goal** function to achieve the following:

1. Robot starts from START position.
2. Follows the line and checks for obstacles.
3. If obstacle is found on a path, avoids it and plans the next shortest path.
4. Stops at GOAL position.

> **Note:** While completing, you have to use the functions defined in **eBot_Sim_Predef.h** header file. In **src/eBot_Sandbox.cpp** file, along with **traverse_line_to_goal** function

header file. In **src/eBot_Sandbox.cpp** file, along with **traverse_line_to_goal** function, some other helper functions are provided. You can complete and use those functions or create your won set of helper functions.

Step-3: Build the project (click on *Project* --> *Build Project*). If there are no errors present in your program, the project will get build successfully and you will get the message (*on console at bottom pane*) as shown below,

```
**========= Build Finished. 0 errors, 0 warnings  ========**
```

Step-4: Check the behaviour of the robot on CoppeliaSim.

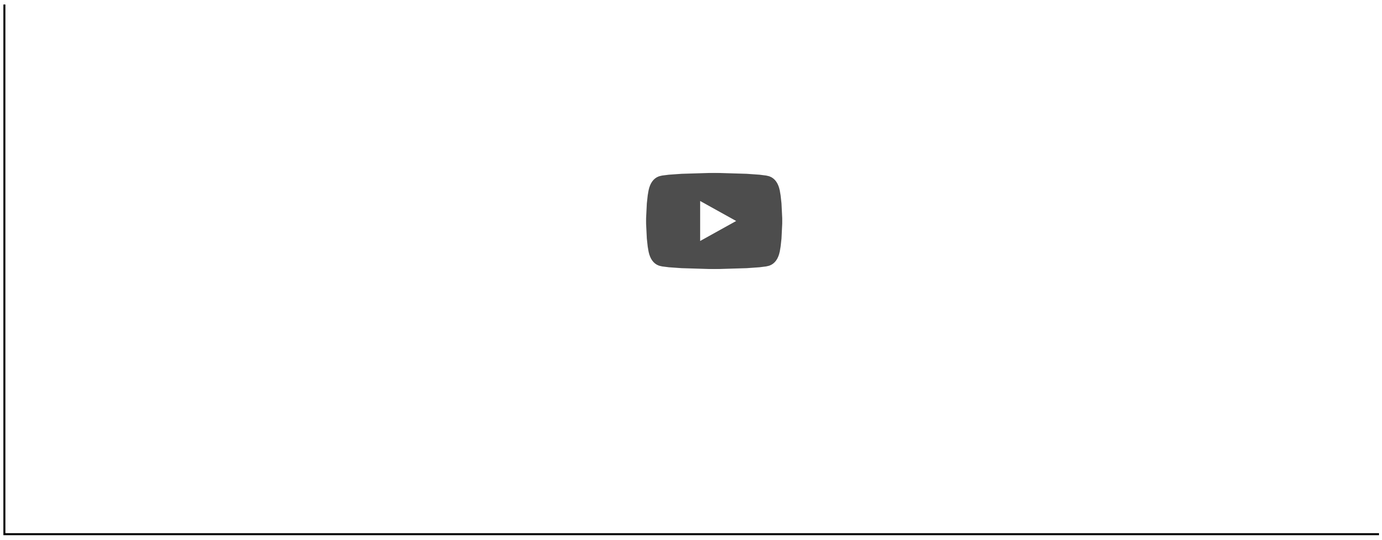
- Download the **lab4_line_follower_robot.ttt** scene. Start CoppeliaSim, click on *File* --> *Open Scene* and browse for **lab4_line_follower_robot.ttt** file to open the scene (shown in *Expected Output - CoppeliaSim* video).
- In Eclipse, right click on **lab4_line_follower_robot** folder (Project Explorer - left pane) and select *Run As* --> *Local C/C++ Application*. If no errors are encountered, following message will be displayed onto the console (bottom pane):

```
Connection Success ...
0
0
0

 Please Enter Y to Start Simulation:      Y
```

Enter 'Y' on console. This will start the simulation in CoppeliaSim.

Simulation: EXPECTED OUTPUT

Software required: CoppeliaSim and Eclipse (refer Installation_Instructions provided to you)



Note: Do not hardcode the path in program. Your code should be generic and it should work for any obstacle positions. During evaluation, your code will be tested for different scene - in which arena will be same but obstacles will be placed at different positions.

Following rules will be followed for obstacle placement:

- No obstacle will be placed on curved or zig-zag path.
- Obstacle will always be placed on a line and between two nodes.
- Only three colored (red, green and blue) obstacles will be used.
- Any number of obstacles can be present on the arena.

All the best!

CS684: Embedded System Course

Lab 6: Case Study - Adaptive Cruise Control (Lustre/Heptagon)

Solution

Solution code for this task is available here.

Use **integrate.sh** from heptagon directory to compile and copy the reactive kernel code into project folder.

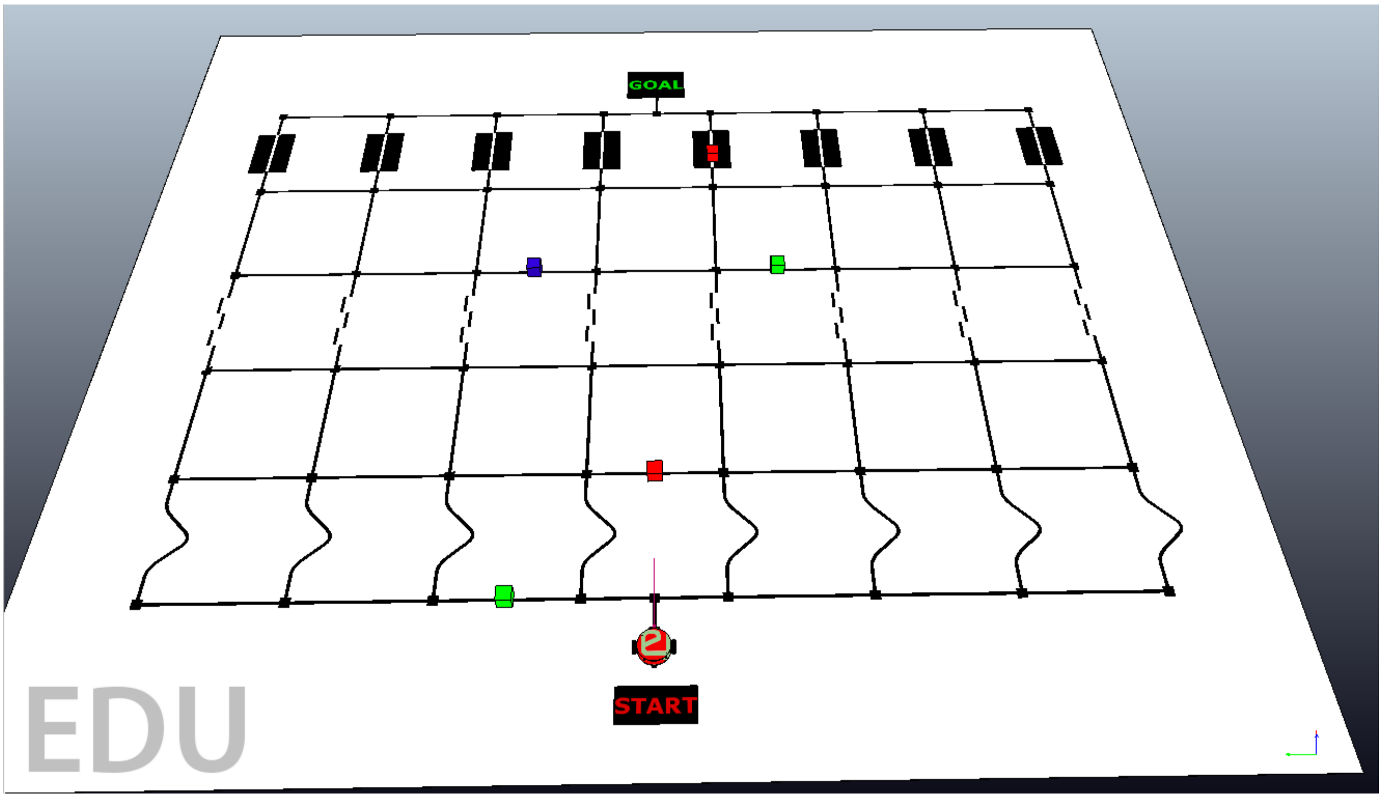
Additional Tips

1. Automatons are great and an exclusive feature. Use them whenever you should. Common scenarios are, you need a **while like** behaviour, you have a big if else ladder conditioned on multiple variables (compare **move** and **moveAut** in the solution) and nested states/multiple modes.
2. Heptagon examples on how to integrate external C code with heptagon.

AIM

In this task, you will learn to implement **Adaptive Cruise Control** using Synchronous Dataflow programming language **Heptagon**.

- The arena provided to you is made up of curved, zig-zag, and straight paths with a START and GOAL position for the robot as shown in following figure.
- Arena also consists of obstacles randomly placed.
- You have to program the robot which will begin from START position, follow the path avoiding obstacles and stop at the GOAL position.



The project folder can be imported as **Eclipse** or **CMake** project.

PROCEDURE

Step-1: Download **lab6_line_follower_robot** zip folder. Right-click on the hyperlinks and select **Save Link As...** option to download. Extract the zip file. Start Eclipse, click on *File* >

Open Projects From File System > *Directory* and browse for **lab6_line_follower_robot** folder to open the project.

Step-2: In **src/supervisor.cpp**, **traverse_line_to_goal** function performs the sense, step and actuate in a loop. You have to complete Heptagon program inside **heptagon directory** which when compiled gives you a reactive kernel implementing the step part. Your Heptagon program implement the following:

1. Robot starts from START position.
2. Follows the line and checks for obstacles.
3. If obstacle is found on a path, avoids it and plans the next shortest path.
4. Stops at GOAL position.

> **Note:** You can write parts of the program in C/C++ if you must. While completing, you have to use the functions defined in **eBot_Sim_Predef.h** header file.
>
> There is an **integrate.sh** file in heptagon directory which you can run each time you change heptagon code to integrate it in the C/C++ project.

Step-3: Build the project (click on *Project --> Build Project*). If there are no errors present in your program, the project will get built successfully and you will get the message (*on console at bottom pane*) as shown below,

```
**========= Build Finished. 0 errors, 0 warnings  =========**
```

> **Note:** The default path to Heptagon C headers is **/usr/local/lib/heptagon/c/**. If you get build errors please include path in Eclipse, specific to your installation.

Step-4: Check the behaviour of the robot on CoppeliaSim.

- Use the **lab6_line_follower_robot.ttt** scene **found in the project folder**. Start CoppeliaSim, click on *File --> Open Scene* and browse for **lab6_line_follower_robot.ttt** file to open the scene (shown in *Expected Output - CoppeliaSim* video).
- In Eclipse, right click on **lab6_line_follower_robot** folder (Project Explorer - left pane) and select ***Run As --> Local C/C++ Application***. If no errors are encountered, following message will be displayed onto the console (bottom pane):

```
Connection Success ...
0
0
0

 Please Enter Y to Start Simulation:      Y
```

Enter 'Y' on console. This will start the simulation in CoppeliaSim.

Simulation: EXPECTED OUTPUT

Software required: CoppeliaSim and Eclipse (refer Installation_Instructions provided to you)

Note: Do not hardcode the path in program. Your code should be generic and it should work for any obstacle positions. During evaluation, your code will be tested for different scene - in which arena will be same but obstacles will be placed at different positions.

Following rules will be followed for obstacle placement:

- No obstacle will be placed on curved or zig-zag path.
- Obstacle will always be placed on a line and between two nodes.
- Only three colored (red, green and blue) obstacles will be used.
- Any number of obstacles can be present on the arena.

All the best!

Rulebook - Search and Rescue

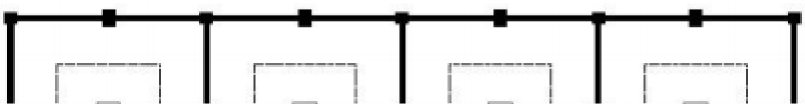
CS684: Embedded System Course

Contents

CS684: Embedded System Course

Rulebook: [1] Theme Description

- Figure 1 shows the arena design for this theme.
- The arena is an abstraction of a disaster-affected area made up of a grid with the **START** position marked.
- The grid is made up of **16 squares** called **Plots**.



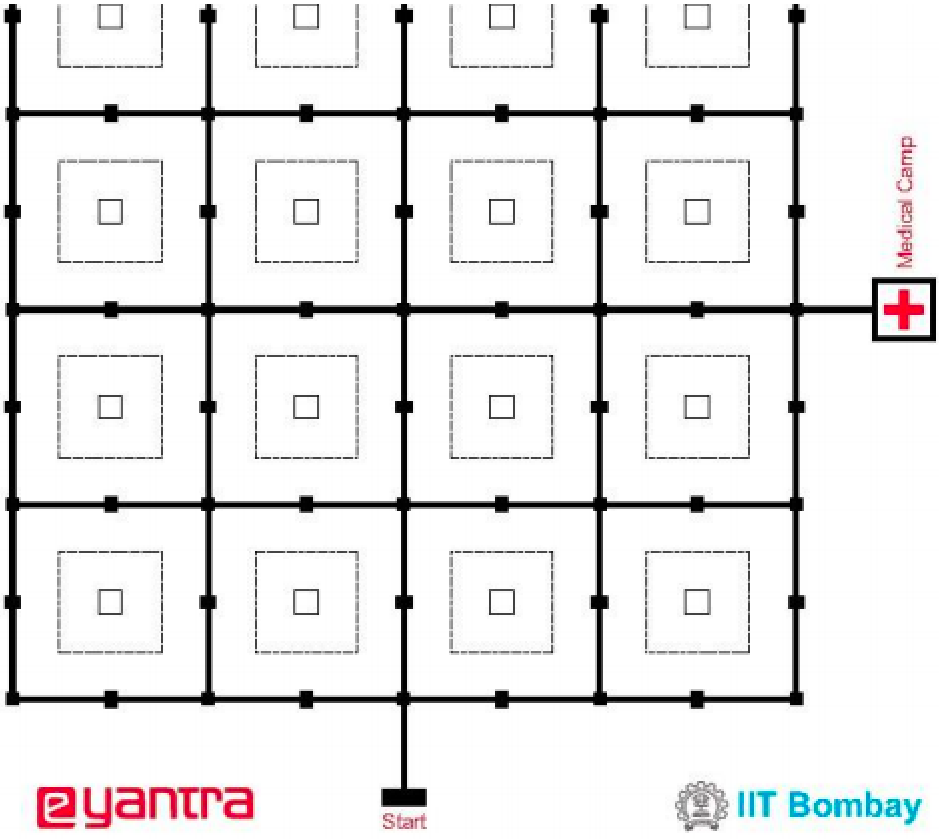


Figure 1: Arena design

---

1. Arena Components

- Each Plot has the following terms associated with it:
  - **Four Mid-Point Markers** on every path around the Plot. Figure 2a highlights the Mid-Point markers for a Plot.
  - The **Clearing Zone**, which is shown by the dotted square of **26cm x 26cm**, highlighted in the green box in Figure 2b.
  - A **6cm x 6cm Inner Square** that is highlighted in Figure 2c.
  - **Four Nodes** present at the corners of every Plot. This is shown in Figure 2d.

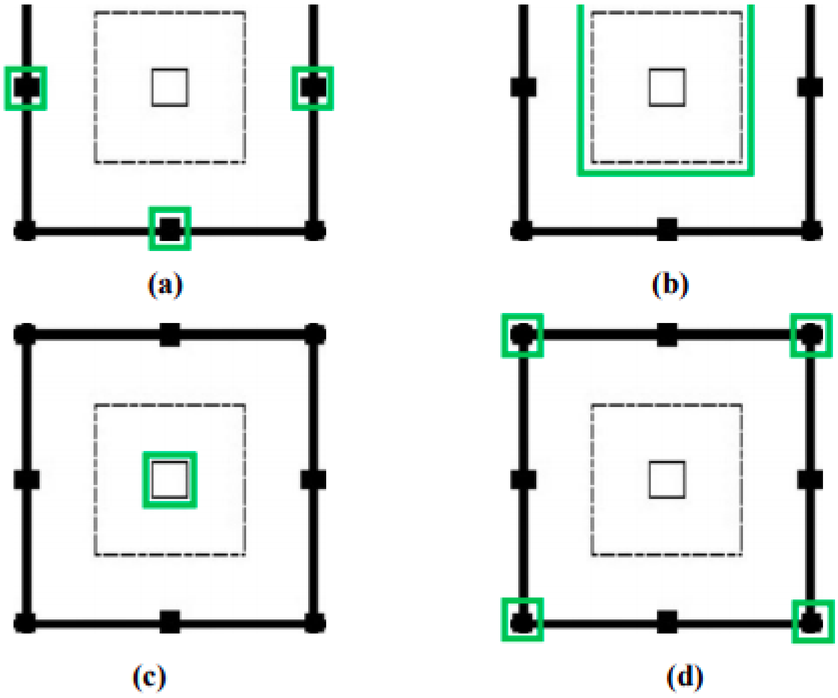


Figure 2: Terms associated with Arena

- Rectangular chart paper patches are used to represent **injured surivirors** and **debris**.
- Three colored patches:
  - **Red** - represent **Survivors with Severe Injuries**, those require **urgent assistance**
  - **Green** - represent **Survivors with Minor Injuries**, those require **not so urgent assistance**
  - **White** - represent pieces of **Debris** strewn on the roads which cannot be moved, thus causing a roadblock

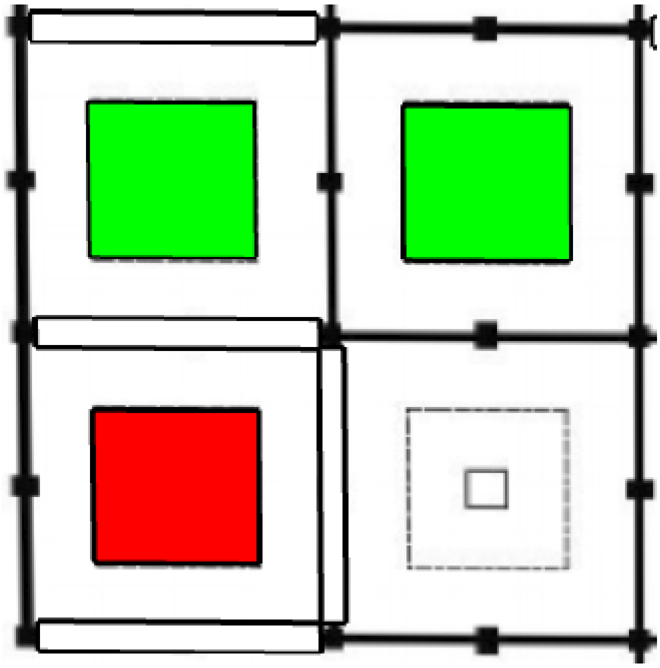


Figure 3: Injured Survivors and Debris on Arena

2. Theme Run Sequence

- The robot will start from the **START** position of the arena.
- It must traverse around the arena avoiding debris and check for the presence of a Survivors in each plot.
- If Survivor is present in the plot, robot detects the type of emergency (Minor or Severe) using color.
    - Communicates the presence and type of Survivor to Desktop/Laptop.
- During scanning, robot will get requests from the server on regular interval of 45 seconds.
    - Robot satisfies the requirement by traversing to the appropriate plot and ringing a buzzer for 1 second. **Note:** This indication can be done only from one of the Mid-Point Markers associated with that Plot.
- Once robot scans entire grid, it should stop at medical camp to mark an end of the run.

---

3. Communication Sequence

- Robot (Firebird V) will communicate with ESP 32 module over UART using serial communication.
- ESP32 will use Bluetooth Low Energy (BLE) to communicate with the internet-connected laptop. Note that ESP32 won't be connected to the Wifi and hence the internet.
- Laptop will communicate with the Thingsboard server.

4. Theme Requirements

Input Operations

- Thingsboard server will send RPC requests to the laptop on a regular interval of **45 seconds**. We will provide you with a script to mock this behaviour in development.
- Laptop receives the RPC requests using MQTT Protocol in JSON format.
- The requests may contain single or multiple requests (actions to perform). Robot can decide to satisfy the requests or ignore them.

- There can be different types of requests as given below:
  - Fetch RED Survivor in 10s: Robot has to travesrse to the nearest RED Survivor plot and ring the buzzer within 10 second to satisfy the requirement.
  - Identify Survivor at plot 4 in 20s: Robot has to traverse to plot 4, identify the Survior and ring the Buzzer within 20 seconds.
  - No Request: No action needs to be performed

Output Operations

- Once a request is performed by the robot, the response should be sent by the robot to the laptop over BLE and from laptop to the Thingsboard server. (The exact request and response formats are present under resource section (`MockingRPC`) **[Updated April 6]**).
- The data should be sent as **telemetry** and the protocol to use while sending the data back to Thingsboard server will be CoAP. **[Updated April 6]**
- Teams should prepare an interface (desktop app or web app) to show the requests received, indicate operations being performed on the arena and result of the operations. This is an open ended task so use your creativity.

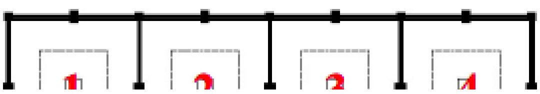
CS684: Embedded System Course

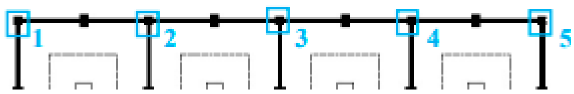
Rulebook: [2] Arena

---

1. Arena Configuration

- For ease of reference:
  - The Plots in the arena are numbered from **1** to **16** as illustrated in red in Figure 1a.
  - The Nodes in the arena are number from **1** through **25** as illustrated in Figure 1b.





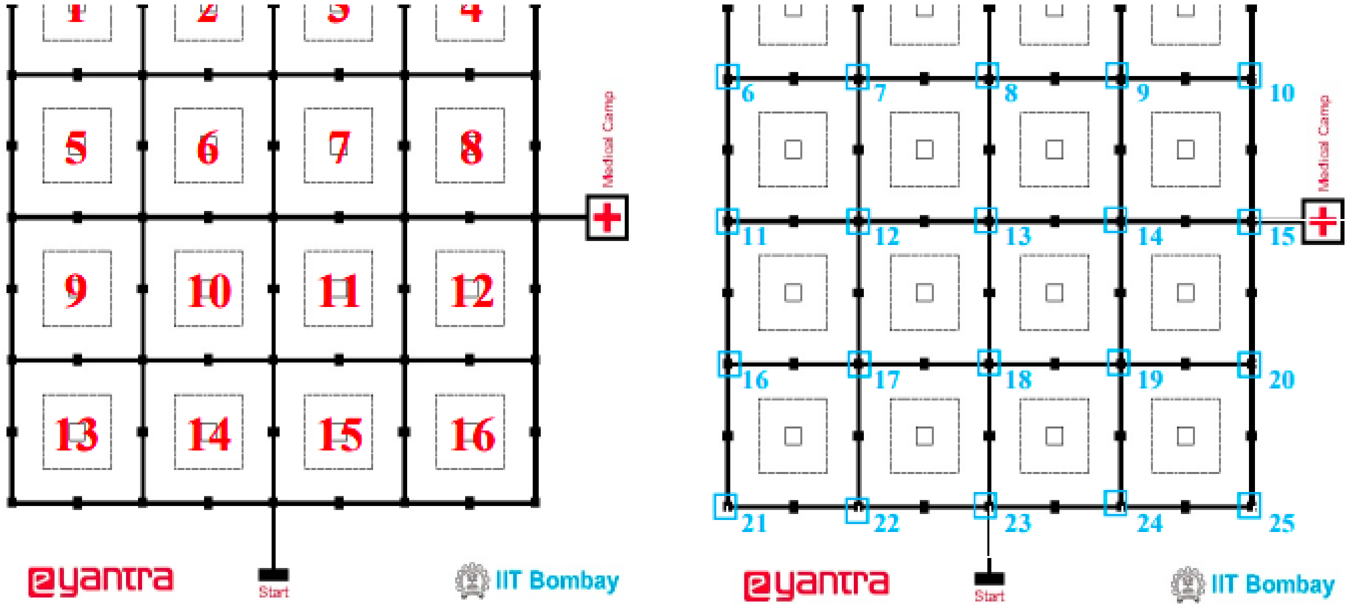


Figure 1: Arena design

---

2. Preparing the Debris and Injured Survivors

- Material required:
  - Red, Green and White color chart paper
- Preparing Debris and Injured Survivors:
  - Teams will prepare three kinds of chart paper patches. Characteristics of these are given in Table 1.

Type of Patch	Length (cm)	Width (cm)	Color	Count
Debris	40	6	**White**	10
Survivors with Major Injuries	26	26	**Red**	10
Survivors with Minor Injuries	26	26	**Green**	10

Table 1: Characteristics of Debris and Injured Survivors

- After preparing these patches, teams must paste them on the arena according to the final arena setup given below.

---

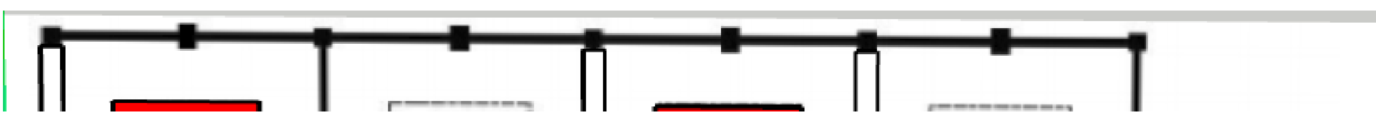
3. Placing the Debris and Injured Survivors on Arena

- Placement of Debris:
  - White patches of size 40cm x 6cm are used to indicate the Debris.
  - These patches need to be sticked on the Path between two nodes and covering Mid-point Marker.
- Placement of Injured Survivors:
  - Green and Red patches of size 26cm x 26cm are used to indicate minor and severly injured survivors.
  - These colored patches need to be sticked in the clearing zone.

Note: You must use transparent sellotape to stick these patches.

---

4. Final Arena Setup



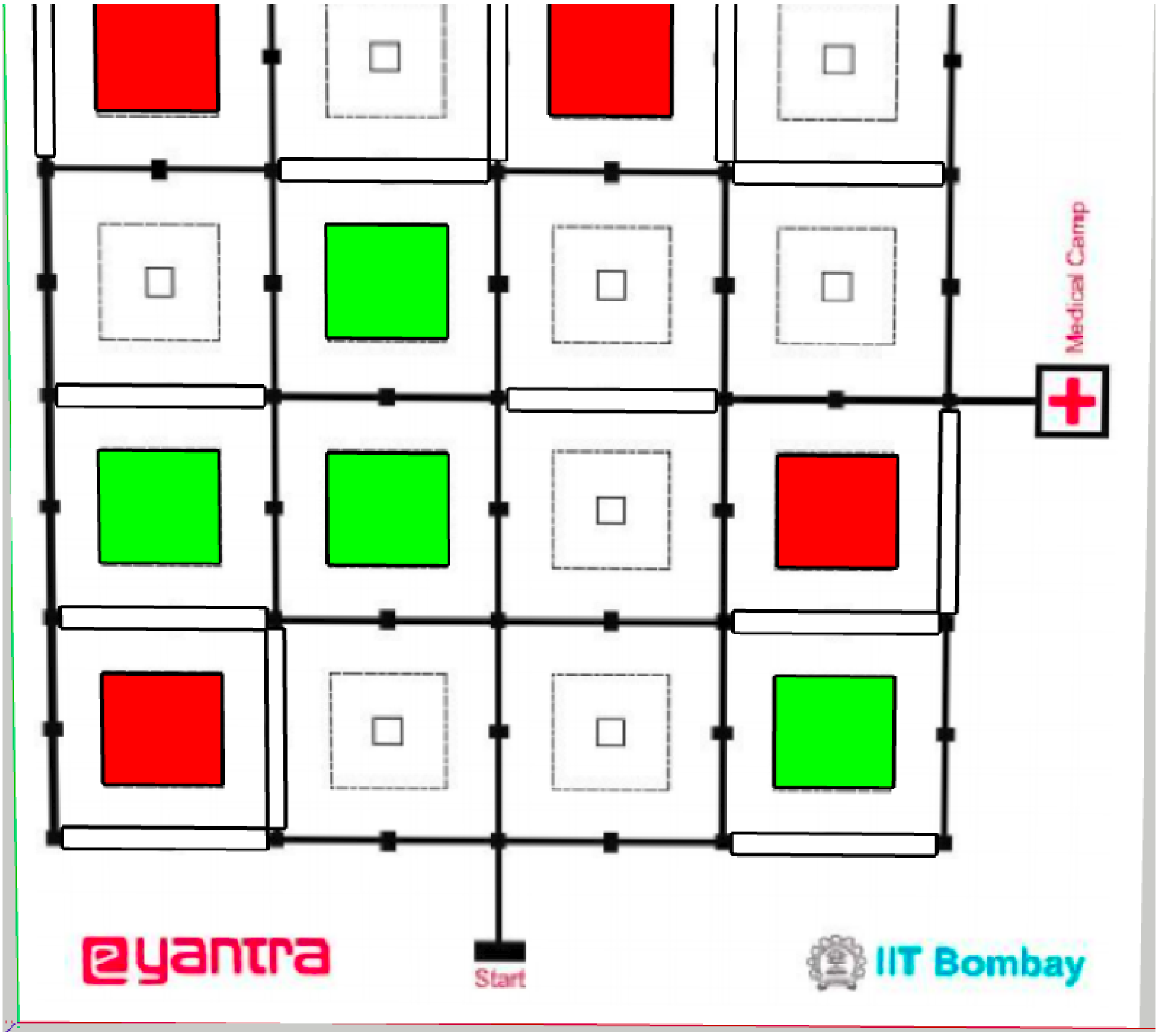


Figure 2: Arena Final Setup

Note: The position of Debris and Survivors is random. During final demonstration, position will be different. Your algorithm should be generic.

CS684: Embedded System Course

Rulebook: [3] Hardware and Software Specifications

---

1. Hardware Specifications

- **Use of Firebird V and ESP32:**
    - All teams must use Firebird V robot and ESP32 development board provided by e-Yantra.
    - Team shall not dismantle the robot.
    - The robot should be completely autonomous. The team is not allowed to use any wireless remote or any other devices such as a camera while the robot is performing the task. The robot are only allowed to communicate using the Wireless Protocol mentioned.
- **Use of additional components:**
    - Firebird V and ESP32 communicate using UART. No other microcontroller-based board shall be attached to the Firebird V robot except ESP32.
    - Teams are not allowed to connect external actuators or structural hardware to the Firebird V robot.
    - Teams are not allowed to use any additional sensor apart from ones provided with the Robot.
- **Power Supply:**
    - The robot can be charged through battery or auxiliary power supply.
    - The team cannot use any other power source for powering the robot.
    - The team can use auxiliary power during practice but the final demonstration should only be made using only battery powered robot.

---

2. Software Specifications

- Teams can use Eclipse, for writting code for AVR microcontroller. Teams are free to use any other open source Integrated Development Environment (IDE) for programming AVR microcontroller.
- Use of any non-open source libraries is not allowed and will result in disqualification.
- Teams can use any language and open source libraries for creating an interface (desktop app or web app).

CS684: Embedded System Course

Rulebook: [4] Theme Rules

Final Run

- The maximum time allotted to complete the task is 10 minutes.
- A maximum of two repositions (explained below) is allowed for each team.
- Once the robot is switched on, human intervention is NOT allowed.
- Robot is not allowed to traverse through the Plot, it always has to follow the black line for traversal.
- For the final demonstration, the Arena Configuration will NOT be given to any of the teams.The robot must navigate through a randomly setup arena and autonomously detect the White Debris, identify the Survivors without any prior knowledge of the arena configuration.
  Note: You MUST have a generic solution that can handle any setup, in real-time.
- Any of the 16 Plots may contain survivor.
- Debris may be placed on path of any of the Plots such that every Plot in the arena will have at least one path for the robot to reach to the Survivor.
- A run ends and the timer is stopped when:
  - The Robot stops at medical camp or
  - If the maximum time limit for completing the task is reached or
  - If the team needs repositioning but has used all repositioning options.

Repositioning the Robot

Suppose while traversing the arena, the robot strays off the black line, team member can place the robot on the previous node (node already traversed by the robot) by dragging (not lifting) the robot back to the line in such a way that both the wheels of robot are parallel to the node and castor wheel is on the black line. This is termed as a Reposition. Note that the timer used for measuring the task completion time in the competition will be continuously running during a Reposition and robot will not be switched off.

CS684: Embedded System Course

Rulebook: [5] Judging and Scoring System

$$Score = (600 - T) + (30 * CDS) + (\sum_{i=1}^{n}(GT_i - TT_i)) * 25 + (10 * NR) + AB - ($$

Parameters:

- T : Time taken by robot to complete the Task
- CDS : Number of Survivors detected correctly
- GT_i : Given time for the ith request
- TT_i : Time taken by robot to complete the ith request
- NR : Number of requests satisfied by the robot
- AB : Bonus of 100 is provided based on the creativity shown in designing desktop/web app
- P : Penalty is incurred each time robot dashes against Debris and for each Reposition.