Lab 1: Software Design and Implementation
Basic Finite State Machines

Objectives
In this lab, you will
i. Communicate software design using finite state machines
ii. Map software design to code using explicit control structures
iii. Create software that has qualities of readability, maintainability and easily upgradability.

Overview
In this lab, you will work in teams to design and implement controllers for simple mechanical systems. Controllers like these are often described in terms of Finite State Machines (FSMs), so the basics of designing FSMs will be described.

This first lab will concentrate on the core elements of FSM implementation within RobotC programs. The use of VEX peripherals, such as push button controls, rotation sensors, motors and timers, can be controlled by RobotC code.

Note: Since RobotC comes with a fairly good debugger, and you have your VEX kits already, feel free to test your code outside of lab time.

Design Details
We can model a simple elevator system as follows:

The building has two floors:
- location = {Ground Floor, First Floor}

On each floor, labeled 0 and 1, there are call buttons with lights
- call buttons: CB0, CB1 = {true, false}
  - These call button flags maintain a true state value after the button is pushed until the button state flag is explicitly set to false. This gives the system memory of floor summon signals until they are explicitly handled.
- call lights: CL0, CL1 = {on, off}

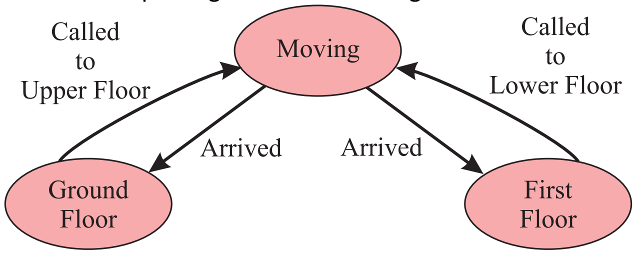
The elevator is moved by a single motor.
- Motor Control: motor = {off, moving up, moving down}

In the history of computer control, many methods have been developed within the Software Engineering community for describing the desired behaviour of automatic control system components including sequence diagrams, flow charts, use case diagrams, etc. If you choose to pursue training as a Computer, Electrical or Software Engineer you will learn much more about these descriptive techniques. Here, we will concentrate on the use of Finite State Machines (FSMs) to describe simple control systems.

A FSM is a mathematical construct of an object which can exist in one of a finite number of states. The object can only be in one of these states at a time with the transitions between these states, the conditions that triggers these transitions, and the outputs in each state describing the behaviour of the object. The objective of the FSM drawing is not to show the flow of logic within the computer code but to show a possible behaviour of a system so that the expected software design can be performed in conjunction with the design of the other components of a system. The FSM specifies what sensors is the software expecting to read and what control signals the software will send in response to these sensor readings. Computer engineering and software engineering students will learn about flow charts and sequence diagrams in later courses for logic flow analysis of algorithms and computer code.

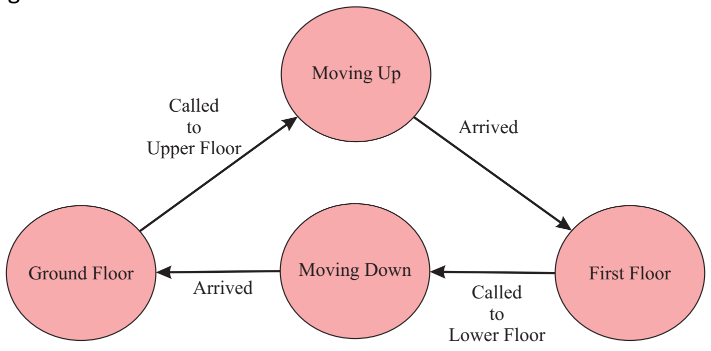
We draw the FSM as a set of nodes with each node representing one possible state of the object. Arrows are drawn on the FSM diagram to represent the possible state transitions of the objects with the label with each arrow indicating the condition that triggers that state transition.

The basic design of the FSM for our elevator example might start off looking like:



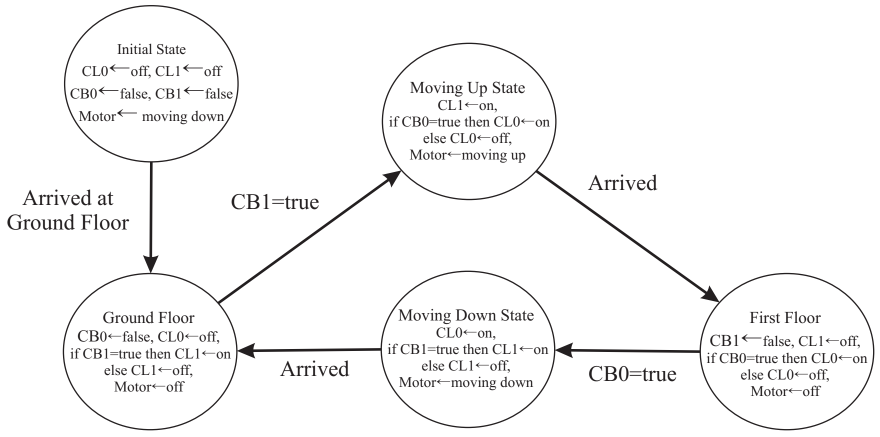
In this case, we have decided that the control system can be in three states: Ground Floor, Moving, and First Floor. All expected transitions are described by the arrows. For example, the arrow labeled “Called to Upper Floor” indicates that when the control system is in the Ground Floor state and the system receives a “Called to Upper Floor” signal, the system transitions to the Moving state.

A good FSM should include all of the desired behaviour of your system and be easily mapped to the computer code which will implement that behaviour. You might decide you want to use states to keep track of which direction the elevator is moving to simplify the motor control mapping in the RobotC code:

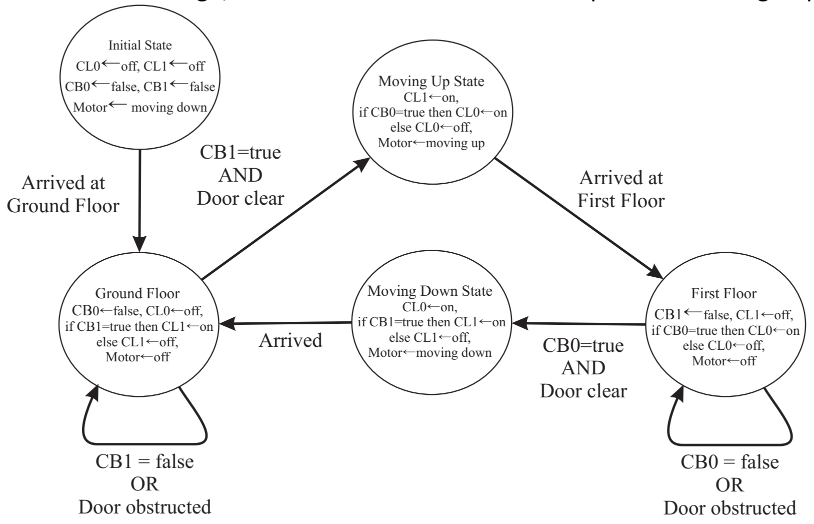


Splitting the states up like this will allow you to have a simple mapping between the states and the required control outputs to the motor. The initial FSM design describes the core elements of your system and their relationships. As you work through the iterations of the design, you may decide to add or delete states to better represent the behaviour of your code. Revisit your original design, if you think of a way to improve it or if it makes the implementation easier.

Once you are happy with the states and transitions within your FSM diagram, you should add the outputs from the control system at each state to the corresponding state node. There are two kinds of FSMs depending on whether the outputs of the FSM are completely determined by the FSM state or not. In Moore FSMs, the output of the FSM is solely determined by the current state. In Mealy FSMs, the output of the FSM is a function of both the state and the current inputs. Below, we can see a FSM diagram of the Mealy FSM for our elevator controller; the value of the CL0 output when in the “Moving Up State” depends on the value of the CB0 input so this is a Mealy FSM.



To refine your FSM, you should think of all of the possible input values received by your control system and what the proper response should be to each input value. Here, we are adding some checks to consider safety considerations with the door and elevator movement. This is far from a final design, but this shows how a FSM can represent evolving requirements.



FSM Implementation

A good FSM design is essential to building a control system but the control system is implemented in computer code. In this course, you will implement your control systems in RobotC and they should correspond to your FSM designs. Linguistic constructs available in RobotC[1] such as: constants[2], enumerated data types[3], and functions will make FSM implementation easy.

Logic

The *Finite State Machine* (FSM) should be a key structural implementation element within your main task. Details should be handled by helper functions. Data flow to and from these functions should be handled by parameters and return types. The core structure of a FSM in RobotC will look something like the code below:

```
while(true) {
  switch( <current state> ) {
    case( <state 1> ):
```

[1] For more information about RobotC, see http://carrot.whitman.edu/Robots/notes.pdf
[2] For an example using const, see http://www.robotc.net/support/nxt/media/sample_programs/NXT/PCF8754.html
[3] For a discussion on typedef enum see http://www.robotc.net/forums/viewtopic.php?f=33&t=1237

```
      // Code executed for state 1 goes here…
      break;
    case( <state 2> ):
      // Code executed for state 2 goes here…
      break;
    // Additional code for other states here
      default:
        //invalid state!!!
    } // switch(  <current state> )
  }  // while true
```

Note that this control structure is an infinite loop! Note also that all control flow should be handled by appropriate control flow mechanisms such as `if` or `switch` statements.

Data Types and Structures

RobotC allows you to organize data into enumerated data types and structures. These mechanisms further accomplish traceability between design and implementation of your system. As a naming convention in the code below, the identifier name for an `enum` or a `struct` starts with "T_" to signify the definition of a new data type:

```
//FSM states
typedef enum T_elevator_state {
       GROUND = 0,
       FIRST  = 1,
       MOVING = 2
};
```

Deliverables:

By the end of this lab, you will need to:

i. Write some RobotC functions to implement some requested behaviours for a simple controller.
ii. Draw a FSM diagram for one such simple controller with some requested behaviour.

Lab Preparation (4 marks)

Before starting, you will need to download firmware onto your controller.

1. Double click on a RobotC icon to get the RobotC software to start. Make sure that If you have any questions, talk to your lab TA.
2. After getting the firmware downloaded to the VEX controller, go to the course webpage and download the skeleton code file named "ProgrammingLab1.c".

Exercise 1 – Push Buttons (2 marks)

Set up two push button sensors and one motor. The first button should cause the motor to start running and the second button should turn the motor off.

Instructions:

1. Setup the **motor** and **push button** in "**Robot→Motors and Sensors Setup"**
   - Motor setup is under the **Motors** tab
   - Connect the motor to the VEX controller via a motor controller unit if you are connecting it to the VEX controller using Motor Port 2-9. The 2-wire motors can be directly connected to the VEX controller with Port 1 or 10.
   - Sensor setup is under the "**VEX 2.0 Analog Sensors 1-8**" and "**VEX Cortex Digital Sensors 1-12**" input tab
   - Use "**Touch**" as the type for the push button, this is available under the "**VEX Cortex Digital Sensors 1-12**" tab.
   - Connect the buttons to Digital ports of the VEX Controller.
2. Write your code in the space under "exercise_1()" in the skeleton code.
   - When the first button is pressed, the motor should be turned on at speed 50
   - The motor should stay running until the second button is pushed.

TESTING YOUR CODE:

1. Before compiling, you MUST save, or changes will not take effect
2. To compile your code, go to "**Robot→Compile Program**"
3. To compile your code AND run it on the VEX controller, go to "**Robot→Compile and Download Program**". If your code fails to compile, fix the reported errors and try again
4. When your code compiles and downloads, you will be taken to the debugger

Debugging

1. Once in the debugger, go to "**Robot→Debugger Windows".**
2. Check that the Debug Windows you want to use are open. You'll usually want to have sensors checked.
3. To run your code, hit "**Start**" or "**Step Into**" on the Program Debug window.
4. The debugger should be in continuous refresh mode by default, but you can change that to run one line at a time by clicking the "**Once"** button in the Debug window
5. You can also stop the code at any point in your program by setting breakpoints. To set a breakpoint, click in the grey bar to the left of the line of code where you want the debugger to stop. A red circle should appear. Click again to remove the breakpoint.
6. To leave the debugger and return to the editor, just close the Program Debug window.

STOP! ***Show a TA your working code before proceeding, and have it marked.***

Exercise 2 – Timer (2 marks)

In this exercise, pushing button 1 will cause the motor to turn on for 3 seconds.

Instructions:

1. Change the #define called EXERCISE_NUMBER to the value 2
2. Write the code to implement the desired behaviour in the function named "exercise_2".
   - Monitor the system for the first button push
   - If the motor is already running when the button is pushed, ignore the input.
   - Otherwise, reset timer T1 and monitor it until 3 seconds have passed (instructions for timing are located in the RobotC left-hand sidebar under "timing")
   - After 3 seconds have passed, turn the motor off and return to monitoring the system for input
3. When you are debugging, go to the **Robot->Debug Windows** and select **timers** and/or **sensors** to see what the timer and sensor values are at any given time during code execution.

STOP! ***Show a TA your working code before proceeding, and have it marked.***

Exercise 3 – Quadrature Encoder Sensor (4 marks)

In this exercise, you are programming your controller to turn the motor shaft 360 degrees when button 1 is pushed.

Instructions:

1. Change EXERCISE_NUMBER to the value 3.
2. Setup a **Quadrature Encoder** under Motor and Sensors Setup. Connect a Motor 393 to your VEX controller. These motors have a big block on their backs which can be connected to the VEX controller's I2C port via a four colour ribbon cable. Under the Motors setup in the RobotC program, set the motor type to "Motor393" and select "I2C_1" for the "Encoder Port."
3. Be sure to hold the sensor and motor stationary while your code is executing!
4. Write your code in the function called "exercise_3" so that, instead of the motor turning off after three seconds as you did in exercise 2, it turns off after the rotation sensor reaches the value 627 (this corresponds roughly to a full rotation of the motor).
   - The Encoder is connected to the I2C port of the VEX controller. Your encoder may count up instead of down when your motor is running depending on the direction.
   - Information on how to use the encoder module is found under the Integrated Encoders sub-section under the Motors help entry in the RobotC GUI.
   - To read a quadrature encoder use the 'getMotorEncoder(m1)' statement where 'm1' is the label assigned to the motor. A quadrature encoder can be reset with 'resetMotorEncoder(m1) ' which sets the current position to be read as angle zero.

STOP! ***Show a TA your working code before proceeding, and have it marked.***

Exercise 4 – Finite State Machine Behavior (18 Marks)

This exercise is slightly trickier. The system should have the following behavior:
If the motor is OFF when input received:

- Button 1 turns motor FORWARDS at power level 50 for approximately 500 points of rotation
- Button 2 turns motor BACKWARDS at power level 50 for approximately 500 points of rotation

If the motor is going FORWARDS when input received:

- Button 1 does nothing
- Button 2 turns motor BACKWARDS for about 500 points of rotation at power level 50 AFTER the current motion has finished.

If the motor is going BACKWARDS when input received:

- Button 1 turns motor on FORWARDS for about 500 points of rotation at power level 50 AFTER the current motion has finished.
- Button 2 does nothing.

In other words, you need to construct a finite state machine with 3 states, where the state of your motor determines what state your system is in: STOPPED, FORWARDS, or BACKWARDS.

Instructions:

1. Sketch an FSM diagram that implements the desired behaviour.

STOP! ***Show your FSM diagram to a TA for marking before proceeding with the code. (5 marks)***

2. Change the EXERCISE_NUMBER to 4 in the skeleton code.
3. Write the code to implement the desired behaviour in the function "exercise_4()".

STOP! ***Show a TA your working code for marking. (8 marks)***

4. Submit code on CourseSpaces website.
    - The filename of your code should be "B0X GNN Lab1.c" with the same substitutions as the subject line.
    - It would be wise to send a copy of your code via email to yourself for safety.

Your code will be marked for readability and functionality. (5 marks)

END OF LAB