# Lab 2: 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 acceptable qualities of readability, maintainability and easily upgradability.

## Overview

In this lab, you will work in teams to design and implement controllers for simple electro-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

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 in advanced courses. Here, we will concentrate on the use of Finite State Machines (FSMs) to describe simple control systems.

A FSM is a mathematical model 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. The behaviour of a finite state machine system is described by the set of possible states, the set of possible transitions between states, the conditions that trigger the state transitions, and the outputs of the finite state machine in each state. A FSM drawing is a compact way of presenting this description. The objective of the FSM drawing is not to show the flow of logic within computer code but to show the behaviour of a software system so that the expected software design can be performed in conjunction with the design of the other components of a system such as mechanical and electrical circuit components. The designers of these other components should be able to quickly determine from the FSM diagram what actions a given software system will take in response to a given stimulus. The FSM specifies what sensors the software will read and what control signals the software will send in response to these sensor readings. Computer, electrical, and software engineering students will learn about other methods of diagraming software systems such as 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 software 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. Inside each node, a label for the state is written along with the output of the software system in that state. This output can either be a fixed output of the inputs to the system as well as the system state.

Let us say that we wished to design the control system for an elevator in a two-storey building. The elevator system can be modelled using the following:

The building has two floors:

- location = {Ground Floor, Second Floor}

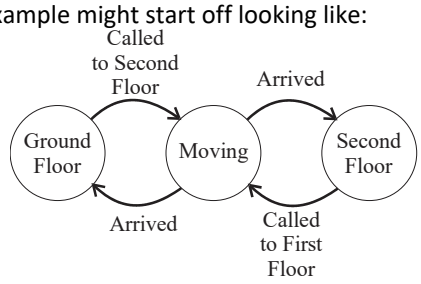
On each floor, labeled 1 and 2, there are call buttons with lights

- call buttons: CB1, CB2 = {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: CL1, CL2 = {on, off}

The elevator is moved by a single motor.

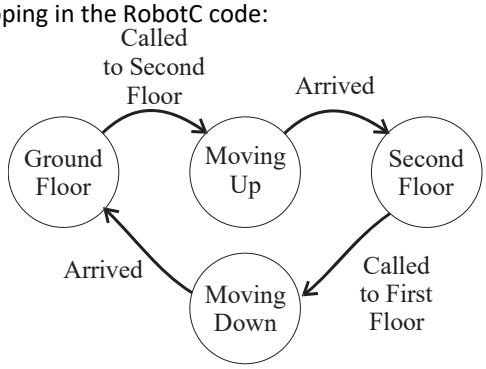
- Motor Control: motor = {off, moving up, moving down}

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



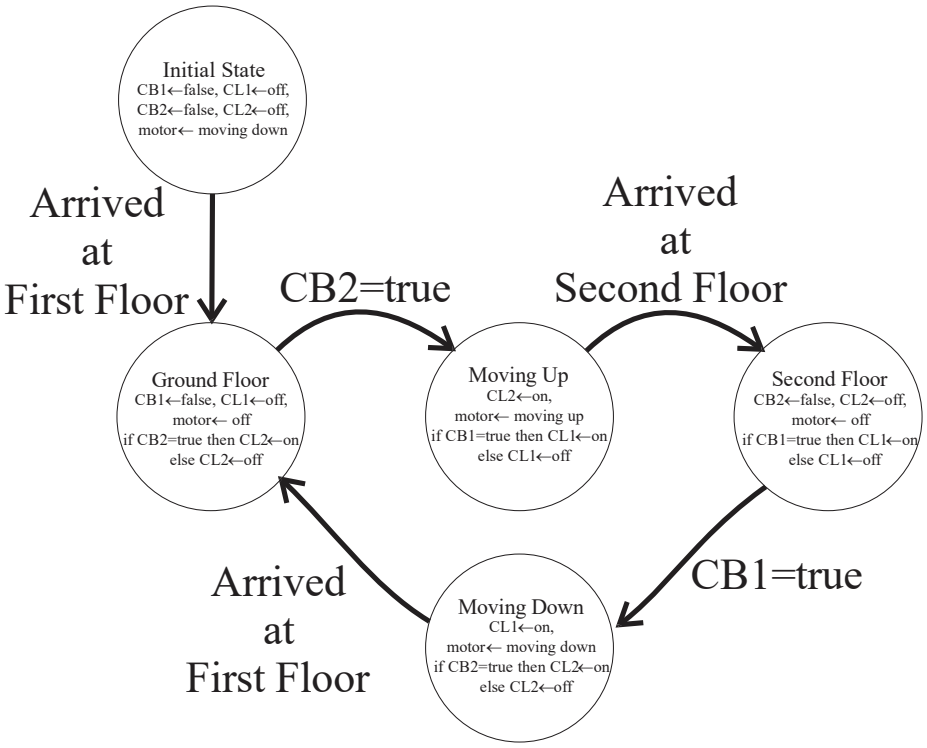

In this case, we have decided that the control system can be in three states: Ground Floor, Moving, and First Floor. All expected transitions are described by the arrows. For example, the arrow labeled "Called to Second Floor" indicates that when the control system is in the Ground Floor state and the system receives a "Called to Second Floor" signal, the system transitions to the Moving state. This FSM diagram is missing the indication of the output of the FSM in each state. In particular, while it is clear that in the "Moving" state that the FSM is directing the motor to operate, it is not clear how the system determines if the motor is to move the elevator up or down while FSM is in 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. The motor control direction using the FSM above is a function of the current state and what floor the elevator is being called to. To make the behaviour clearer, we might decide to use states to keep track of which direction the elevator is moving to simplify the motor control mapping in the RobotC code:

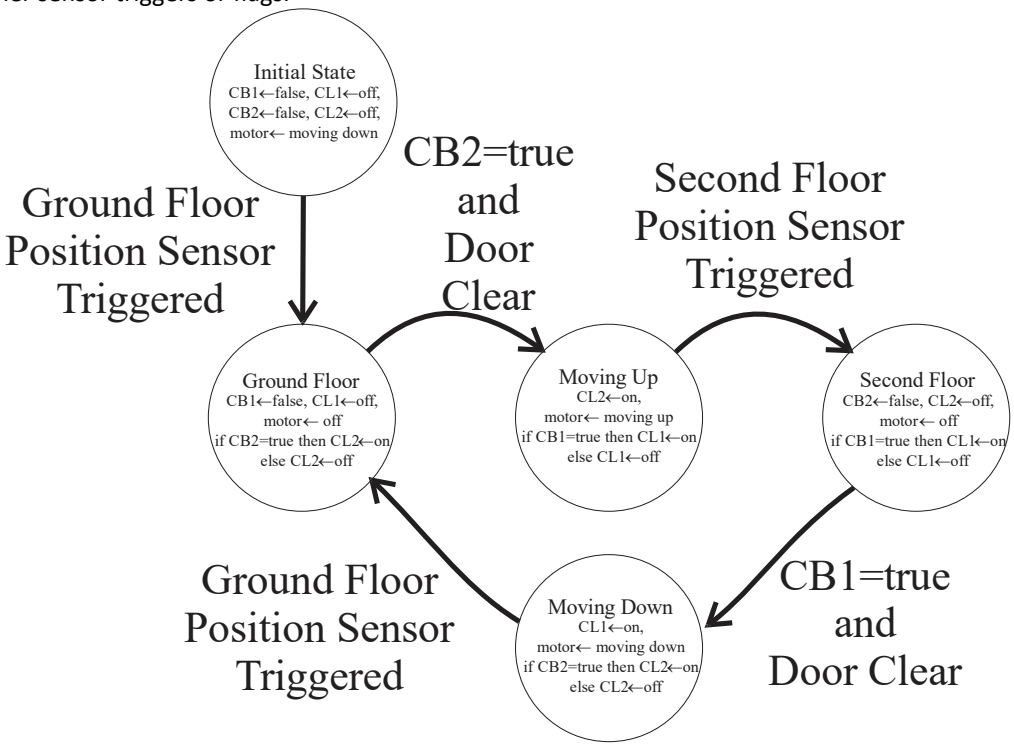



Splitting the states up like this will allows us to have a simple mapping between the states and the required control outputs to the elevator motor. The motor direction for the second FSM is a simple one-to-one mapping from the state. As you work through the iterations of a design, you may decide to add or delete states to better represent the behaviour of your code. A FSM not only represents the software design but also provide a way of presenting the behaviour of your system to people using it. It should be easy for a person without software experience to look at your FSM diagram and determine how your system will behave. The FSM should be adjusted, if you think of a way to improve behaviour or if the change makes the mapping to 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 CL1 output when in the "Moving Up State" depends on the value of the CB1 input so this is a Mealy FSM. In the FSM diagram below, the equal sign '=' is used to indicate a check for equal value between two items. When we want to assign a value to an item we use the arrow symbol '←' so that 'CL1←on' indicates that we are assigning the value of 'on' to CL1 (we are turning on the call light for floor 1). Note that we have added a special Initial State so the system goes to a known state upon start-up.




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. Note that state transitions are triggered by either sensor triggers or flags.



## 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 help to 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> ):
      // 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` starts with "T_" to signify the definition of a new data type:

```
//FSM states
// Define enumerated state with tag 'T_elevator_state' with three possible values
enum T_elevator_state {
       GROUND = 0,
       FIRST  = 1,
       MOVING = 2
};
```

You can use this new type much like a regular variable type in your code:

```
// Declare variable of type T_elevator_state
T_elevator_state elevator_state = GROUND;
```

You can also use these types in logical expressions:

```
if ( elevator_state == GROUND ) {
  // Code here will only execute when elevator_state is equal to GROUND
  …
}
```

[1] For more information about RobotC, see http://www.stormingrobots.com/prod/tutorial/RobotCPacket I.pdf
[2] For an example using const, see http://www.stormingrobots.com/prod/tutorial/RobotCPacket II.pdf
[3] For a discussion on typedef enum see http://www.robotc.net/forums/viewtopic.php?f=33&t=1237

## 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. Read this laboratory description. Be prepared to answer questions about the material above.
2. Double click on a RobotC icon to get the RobotC software to start. If you have any questions, talk to your lab TA.
3. After getting the firmware downloaded to the VEX controller, go to the course webpage and download the skeleton code file named "ProgrammingLab2.c".
4. Draw a Finite State Machine diagram for the system which implements the required behaviour for Exercise 3 below.
5. Write a first draft of the code for Exercise 2 in the "exercise_2()" section of the code file.
6. Write a first draft of the code for Exercise 3 in the "exercise_3()" section of the code file.

## Exercise 1 – Push Buttons (4 marks)

Set up two push button sensors and one motor. You are provided with code for this exercise. The first button should cause the motor to start running and the second button should turn the motor off.

**Instructions:**

1. Set up a motor and two push buttons to your VEX controller.
    a. Connect a two-wire motor with an 'Integrated Encoder Module' to port 1 of the 'MOTOR' block on the VEX controller. Look for the text integrated Encoder Module on the back end of the motor. The motor will have a large port with four pins visible on the side. You do not have to connect any cable to this port at this time. The connector for the motor will have two prongs on its connector. This should be connected to port 1 with the tab on one side of the connector going into the appropriate slot in the VEX controller block.
    b. Connect the push buttons into slot 1 and slot 2 of the 'DIGITAL' block on the VEX controller. Similar to the motor connector, the tab on the connector should go into the appropriate slot in the VEX controller's block.
2. Examine the **motor** and **push button** in "**Robot→Motors and Sensors Setup"**
    - The direction "**Robot→Motors and Sensors Setup"** indicates that you should first select the "**Robot**" top menu item and then select the "**Motor and Sensor Setup**" item below that.
    - Motor setup is under the "**Motors"** tab of the "**Motors and Sensors Setup**" window.
    - 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.
3. Examine the code under "exercise_1()" in the skeleton code to see how it operates.
    - 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 THE 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, a "**Program Debug**" window will be opened.

**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 the sensor debugger windows open. These will show you what the current sensor values are.
    - To run your code, hit "**Start**" button on the "**Program Debug**" window.

- To run one line of your code hit the "Step Into" button on the "**Program Debug**" window. This will allow you to see what a single line of code does. You can advance through the code by one line at a time by hitting this button.
- 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

3. Set a breakpoint at the line with the `switch(exercise1_state)` statement in the `exercise_1()` function.
   - Click in the grey bar to the left of the line with the `switch(exercise1_state)` statement.
   - A red dot should appear next to the line in the grey bar.
4. Run the code and see that the execution stops at this line.
   - If the Program Debug window is not already open, select "**Robot→Compile and Download Program**" or hit the "**Download to Robot**" button on the top of the screen.
   - Hit the "**Start**" button in the Program Debug window.
   - Almost immediately, the code should stop executing at the line where the breakpoint was set. When this happens, there should be a yellow arrow on top the red dot on the grey bar.
5. Check the values in the debugger windows.
   - Select "**Robot→Debugger** Windows" to ensure that there are check marks next to 'Global Variables', 'Local Variables', 'Motors' and 'Sensors'.
   - If not select these options under the menu to activate the check marks.
   - You should 'Global Variables', 'Local Variables', and 'Motors' and 'Sensors' tags in the bottom pane of the screen.
   - When you click the 'Sensors' tag, you can see what the current values from the sensors are. Push both buttons and see how the values change in the debugger window.
   - Click on the 'Global Variables' tab and examine the values of the 'button1_pushed' and 'button2_pushed' variables.
   - Click on the 'Local Variables' tab to see the value of the 'exercise1_state' variable. If no variable values show up, click on the 'Step Into' button to go the next line and update the window.
6. Step through the code to see when the 'button1_pushed' flag is updated.
   - Hold down the first button and then step through the code until the `button1_pushed` flag goes to `true`.
   - When the flag is set to `true`, release the button.
   - Continue stepping through the code. Note when the motor is turned on and when the `button1_pushed` flag is set back to `false`.
   - Examine what is visible under the 'Motors' tab when the motor is running and not running.
7. Repeat step 6 above except for second button instead of the first.
8. To leave the debugger and return to the editor, just close the Program Debug window.

**DURING THE LAB:** ***Be prepared to answer the following questions:***

1. ***When are the button1_pushed and button2_pushed flags set to true during code execution?***
2. ***When are the button2_pushed and button2_pushed flags set to false during code execution?***

## Exercise 2 – 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 if the motor is currently stationary. Hitting the button again during the rotation should not cause the rotation to repeat.

**Instructions:**

1. Change EXERCISE_NUMBER to the value 2.
2. Setup a **Quadrature Encoder** under Motor and Sensors Setup. Connect a Motor 393 to your VEX controller. Under the Motors setup in the RobotC program, ensure that the motor type is set to "Motor393" and that "I2C_1" has been selected for the "Encoder Port." 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 that you should have in your kit.
   - Sometimes the VEX controller does not properly initialize an encoder so it is not read correctly. If your motor's encoder is properly connected the light on the back of the motor should give two quick green flashes about every 3 seconds. If it is not addressed correctly, the light will not flash or it will be flashing or solid amber. In this case, power down the VEX controller completely by turning off the power and disconnecting the programming cable followed by reconnecting the programming cable and powering the controller back up to see if this causes the encoder to be properly initialized. If this does not work call a TA over to see if they can help you.

- When you connect a second quadrature encoder, its cable will connect to the I2C port of the first encoder is addressed as "I2C_2".

4. Write your code in the space under "`exercise_2()`" 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 encoder reads 627 ticks or greater. The value of 627 ticks roughly corresponds to one full rotation of the motor.
   - Feel free to copy the code from the "`exercise_1()`" function to use as a starting point.
4. Some tips about writing this code:
   - The Encoder is connected to the I2C port of the VEX controller. 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 of the motor as the zero tick point. Note that the code supplied does not use '`m1`' as the motor label.
   - Make sure that you reset the encoder so that a second push of the button will cause the motor to do a second rotation.

**DURING THE LAB:** ***Show a TA your working code and show that it gives the required behaviour.***

## Exercise 3 – 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 3000 points of rotation
- Button 2 turns motor BACKWARDS at power level 50 for approximately 3000 points of rotation

If the motor is going FORWARDS when input received:
- Button 1 does nothing
- Button 2 turns the motor BACKWARDS for about 3000 points of rotation at power level 50 AFTER the current FORWARDS motion has finished its 3000 points of rotation.

If the motor is going BACKWARDS when input received:
- Button 1 turns motor on FORWARDS for about 3000 points of rotation at power level 50 AFTER the current BACKWARDS motion has finished its 3000 points of rotation.
- Button 2 does nothing.

In other words, you need to construct a finite state machine with at least 3 states, where the state of your motor determines what state your system is in: STOPPED, FORWARDS, or BACKWARDS. You may need additional states to handle the case where a button is pushed when the motor is activated. You will lose marks if the behaviour of your FSM is ambiguous. For example, if it is not clear if your FSM will retain the memory of button pushes made when the motor is active.

**Instructions**:

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

**DURING THE LAB:** ***Show your FSM diagram to a TA. (5 marks)***

2. Change the EXERCISE_NUMBER to 3 in the skeleton code.
3. Write the code to implement the desired behaviour in the function "`exercise_3()`".

**DURING THE LAB:** ***Show a TA your working code for marking. (8 marks)***

4. Submit code on CourseSpaces website.
   - The filename of your code should be "B0X_GNNN_Lab2.c" with the "B0X" being replaced with your lab section and "GNNN" being replaced with your group number preceded by a 'G'. Use of other filenames will prevent your code from being marked.
   - It would be wise to send a copy of your code via email to yourself for safety.
   - Make sure code is easy to read. See the sample code for an example of how clean code should be written.

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

END OF LAB