Lab 0 - Signal Conditioning and Filtering

Report

ECE 118

Elkaim

4/14/24

Andrew Ton That

Aleida Diaz-Roque

Part 1: PCB Assembly and Soldering

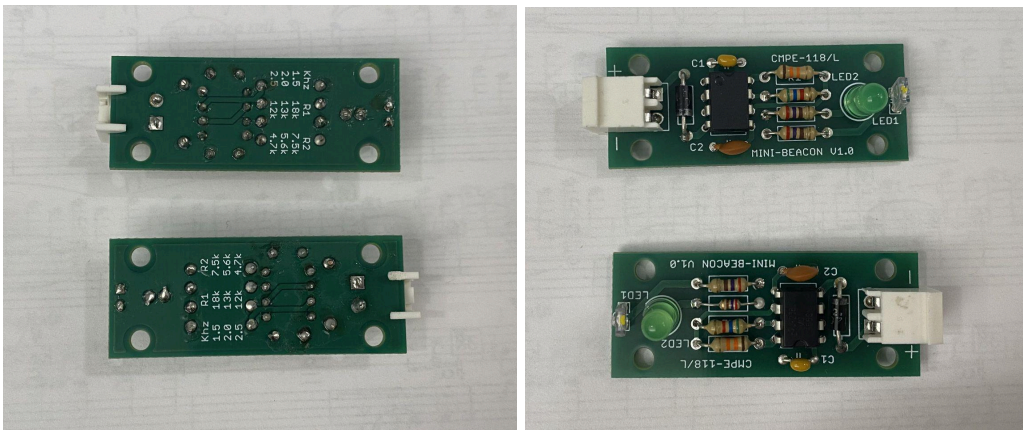

Figure 1. Mini Beacon (Top: Aleida Diaz-Roque, Bottom: Andrew Ton That)

Observations:

Leaded solder seemed to be much easier to work with than the unleaded solder thanks to its lower melting point. However, both types of solders came with difficulties. One of the biggest problems when soldering this board was “cold soldering” where the solder would form a bubble rather than a volcano-like formation around the pin. This was caused by the soldering iron not being used for long enough on each solder point, and was fixed by just reheating the cold solder until it properly formed a strong contact with the PCB and pin.

Part 2: “Hello World!” on a Roach

Getting the new roach board to print “Hello World” was slightly more difficult than it may have first seemed. Following the given instructions for setting up a new MPLabX project was pretty straightforward, but actually getting the roach to print anything required some adjustments that were not stated in the setup document. Mainly, in order to be able to print anything, the project had to be assigned a heap of any size. A size of 1024 was chosen and all

subsequent parts were assigned a heap of size 1024. In addition, since the print statement was not placed inside of a loop of any kind, we suspect that the output of the print statement was caught on some weird character buffer bug where only part of the message would print. To solve this, two consecutive print statements were used and this seemed to fix the issue.

```
int main(void) {
	BOARD_Init();
	Roach_Init();

	printf("Hello World\n");
	printf("Hello World\n");
	while (1) {
	;
	}
	// return 0;
}
```

Part 3: Running the Roach Test Harness

Upon loading the test harness, the robot flashed the LED bar 5 times then flashed alternating lights, just as described in the lab document/test harness documentation. Pressing the front left bumper with the power switch off resulted in the LED bar flashing, but remaining off afterwards. Pressing the same bumper with the power switch on resulted in a voltage reading of 309, and the LED bar remaining on for a longer period.

Pressing the front right bumper flashed the LED bar and resulted in the light sensor values printing to the serial terminal. In addition, the LED bar is illuminated reactively to the amount of light received by the sensor, with a dark environment resulting in a fully illuminated LED bar and full illumination resulting in no LED bar output. A high sensor reading corresponds to a low level of light, and a low sensor reading corresponds to a high level of light

Pressing the rear left bumper flashed the LED bar 3 times then caused the left motor to rotate at various speeds. After driving the motor, the LED bar flashed an additional 3 times. This showed that the motor is functional. Pressing the rear right bumper resulted in an identical process for the right motor, proving that both motors are functional.

Part 4: Roach Hardware Exploration

Our Test Harness is designed to ensure that the motors, bumpers, light sensors, and LED bar work as intended. We used our design for the general inputs/effects from the prelab with a few changes. It was originally intended to use keyboard inputs to drive the motors, but was changed to use bumper inputs instead. Depending on which bumper was pressed, the left or right motors would spin forwards or backwards. The speed of the motors was determined by the light sensor, where the level of light could also be read on the LED bar by the number of lights illuminated.

The difficulty of writing this test harness came from figuring out the functions/commands required to interface with the roach's various inputs and outputs. For this, the roach.c test harness was used as an example of how to drive motors, read sensors, and in general interface with inputs and outputs.

This is part of the code used for the test harness. The switch statement is used to differentiate between which bumper was pressed.

```
switch (Roach_ReadBumpers()) {
            case FLEFT_BUMP_MASK:
            printf("Front Left!\r\n");
            Roach_LeftMtrSpeed(motorSpeed);
            break;

            case FRIGHT_BUMP_MASK:
            printf("Front Right!\r\n");
            Roach_RightMtrSpeed(motorSpeed);
```

```
            break;

            case RLEFT_BUMP_MASK:
            printf("Rear Left!\r\n");
            Roach_LeftMtrSpeed(-motorSpeed);
            break;

            case RRIGHT_BUMP_MASK:
            printf("Rear Right!\r\n");
            Roach_RightMtrSpeed(-motorSpeed);
            break;

            default:
            Roach_LeftMtrSpeed(0);
            Roach_RightMtrSpeed(0);
            break;
      }

      // light level to LED bar
      lightLevel = Roach_LightLevel();
      Roach_BarGraph(lightLevel * 12/1023);
      motorSpeed = lightLevel * 100/1023 + 20;
```

Part 5: Event Detection

Initially we began by using the instructions provided on creating a project using the events and services framework. This guided us through running the simple event checker testing the battery connection. Once we got the event checker to work we were able to use the design of the battery event checker as a model for our two event checkers.

The first event checker we designed was based on the pseudocode made during the prelab which reads the status of the bumpers and then compares the value to 0x00, representing no bumpers pressed. If the values were not equivalent then we determined that a bumper was pressed and if they are equivalent then they are released. Below is a snippet of the logic we used to implement the event checker with only relevant lines included

```
uint8_t CheckBumper(void) {
    static ES_EventTyp_t lastEvent = BUMPER_RELEASED;

    … other lines
```

```
    uint8_t returnVal = FALSE;
    uint16_t bumperValue = Roach_ReadBumpers(); // read bumpers

    if (bumperValue == 0x0) { // are all bumpers released
        curEvent = BUMPER_RELEASED;
    } else {
        curEvent = BUMPER_PRESSED;
    }
    if (curEvent != lastEvent) { // check for change from last time
        thisEvent.EventType = curEvent;
        thisEvent.EventParam = bumperValue;
        returnVal = TRUE;
        lastEvent = curEvent; // update history


      set curEvent to BUMPER_PRESSED

      … other lines

return returnVal
```

The CheckLightSensor function is similar in structure to the CheckBumper function but it is designed to determine if the environment is light or dark. It compares the sensor's reading to a predefined threshold and if the light condition changed from the previous check, the function records the event and signals a change. Below is a snippet of the logic we used. Note that this implementation does not use any hysteresis whatsoever, so the several light events will be triggered at once due to noise.

```
uint8_t CheckLightSensor(void) {
      uint16_t lightValue = Roach_LightLevel();
      uint16_t threshold = 600;

      if (lightValue < threshold) { // lots of light
            curEvent = LIGHT_SENSOR_LIGHT;
      } else {
           curEvent = LIGHT_SENSOR_DARK;
      }

      if (curEvent != lastEvent) { // check for change from last time
          thisEvent.EventType = curEvent;
          thisEvent.EventParam = lightValue;
          returnVal = TRUE;
          lastEvent = curEvent; // update history
```

Debugging this part of the project took the longest since the functions did not change much from our pseudocode. The bumper event checker worked on the first couple of attempts as we had missed minor errors. For the light event checker we struggled to print out a reading for the light levels and get it to cause an event more than once. After a couple hours the only piece that was missing was adding RoachInit() to the main function of the EventChecker.c file. We assumed that since the bumper event checker worked correctly, everything we needed had already been initialized. It was the only "bug" we encountered which had led us to think our design was flawed. In reality adding the initialization solved all our issues.

Part 6: Better Event Detection

To learn more about the structure of simple services within the framework, we followed the instructions for running a test harness having the keyboard as inputs and getting the simple service running using the template. Once we reformatted the naming of the project to make sense for our own, we continued to implement the service and better event checkers.

To implement the "better" event checkers for the switches and solve the debouncing issues we used two methods. The first method involved bit shifting to determine a stable press or release.

```
ES_Event RunService(ES_Event ThisEvent) {

… other lines of code

      uint16_t bumperValue = Roach_ReadBumpers();

      pastValues = pastValues << 1;

      if (bumperValue != 0x0) { // all released
            pastValues = pastValues | 0x01;
      }

      if (lastEvent == BUMPER_RELEASED) { // debnc the press
            if (pastValues & 0xFFFFFFFF == 0xFFFFFFFF) {
```

```
                  curEvent = BUMPER_PRESSED;
            } else {
                  curEvent = BUMPER_RELEASED;
            }
      } else if (lastEvent == BUMPER_PRESSED) { // debnc release
            if (pastValues == 0) {
                  curEvent = BUMPER_RELEASED;
            } else {
                  curEvent = BUMPER_PRESSED;
            }
      }

      if (curEvent != lastEvent) { // check change from last time
            ReturnEvent.EventType = curEvent;
            ReturnEvent.EventParam = bumperValue;

            lastEvent = curEvent; // update history

… other lines of code

}
```

This code implements debouncing for bumper inputs by shifting the state of pastValues left by one bit each cycle and conditionally setting the least significant bit based on the current bumper state. If bumperValue is not 0, it means a bumper is pressed, so 0x01 is OR-ed with pastValues to set its least significant bit. Debouncing is achieved by checking the accumulated history of pastValues. For a press to be recognized as stable (BUMPER_PRESSED), all bits in pastValues must be set (checked against 0xFFFFFFFF). Conversely, for a release to be recognized as stable (BUMPER_RELEASED), pastValues must be 0. This ensures that only consistent states over time are considered as valid events. The event is updated (ReturnEvent.EventType and ReturnEvent.EventParam) and history maintained (lastEvent) only when the current event (curEvent) differs from the last, indicating a stable transition from press to release, or vice versa.

This method triggered events upon press and release but we realized that if another bumper was pressed while one had remained pressed the event would be consumed. This

prompted us to change the method we used to debounce. This new implementation uses an array and does detect multiple events.

```
ES_Event RunService(ES_Event ThisEvent) {

… other lines of code

      // Read Bumper Code
      uint16_t bumperValue = Roach_ReadBumpers();
      bumperValues[index] = bumperValue;

      // Bumper array
      index = (index + 1) % BUMPER_BUFFER_SIZE;

      // check that every value in array is same
      int stable = 1;
      for (int i = 0; i < BUMPER_BUFFER_SIZE; i++) {
            if (bumperValues[i] != bumperValue) {
            stable = 0;
            break;
            }
      }

      if (stable == 1 && bumperValue != prevBumperValue) {
      // check for change from last time
            ReturnEvent.EventType = BUMPER_CHANGED;
            ReturnEvent.EventParam = bumperValue;
            prevBumperValue = bumperValue; // update history
     }

… other lines of code

}
```

It reads the current state of the bumpers using Roach_ReadBumpers(), storing the value in bumperValue. This value is then added to a circular buffer bumperValues, managed by incrementing index with wrap-around. To determine if the bumper state is stable (debounced), it checks if all elements in bumperValues match the latest bumperValue. If they are all the same (stable remains 1), and the bumper state has changed from the previous check (bumperValue differs from prevBumperValue), it is considered a valid state change. The code then sets the EventType to BUMPER_CHANGED, updates the event parameter with the current bumperValue, and records this value as prevBumperValue for future comparisons. This process

ensures that only consistent and stable changes in the bumper state are recognized as events, effectively filtering out transient fluctuations or noise.

To improve the light change detection we implemented the light checking with hysteresis with an upper and lower bound for the change from light and dark. This was modeled after the example in the notes for the roach.

```
#define LIGHT_THRESHOLD 470
#define DARK_THRESHOLD 530

ES_Event RunService(ES_Event ThisEvent) {
      ES_Event ReturnEvent;
      ReturnEvent.EventType = ES_NO_EVENT; // assume no errors

      /****************************************
      in here you write your service code
      ****************************************/
      static ES_EventTyp_t lastEvent = BUMPER_RELEASED;
      static uint16_t prevBumperValue = 0x0;
      // static char pastValues = 0x00;
      static uint16_t bumperValues [BUMPER_BUFFER_SIZE];
      static int index = 0;
      ES_EventTyp_t curEvent;

      switch (ThisEvent.EventType) {
      . . .
     other code
     . . .
      case ES_TIMEOUT:
            ES_Timer_InitTimer(SIMPLE_SERVICE_TIMER, TIMER_0_TICKS);

          . . .
          Other Code
          . . .


            // do light check
            static ES_EventTyp_t lastLightEvent = LIGHT_SENSOR_LIGHT;
            ES_EventTyp_t curLightEvent;
            ES_Event thisLightEvent;
            uint16_t lightValue = Roach_LightLevel();

            if (lastLightEvent == LIGHT_SENSOR_LIGHT && lightValue >
DARK_THRESHOLD) {
            curLightEvent = LIGHT_SENSOR_DARK;
            } else if (lastLightEvent == LIGHT_SENSOR_DARK && lightValue <
LIGHT_THRESHOLD) {
            curLightEvent = LIGHT_SENSOR_LIGHT;
            } else {
            curLightEvent = lastLightEvent;
```

```
            }


            if (curLightEvent != lastLightEvent) { // check for change from
last time
            printf("\r\nProcessed light event in the service instead of event
checker!");
            ReturnEvent.EventType = curLightEvent;
            ReturnEvent.EventParam = lightValue;
            lastLightEvent = curLightEvent; // update history
```

The light checking code was incorporated into the existing service so a light value is sampled at a rate of 200 Hz, the same rate as the bumper. This implementation of the light checker incorporates hysteresis in its logic to prevent frequent toggling between light and dark states due to minor fluctuations in light level readings. Hysteresis is achieved by using two distinct thresholds: `LIGHT_THRESHOLD` for transitioning from dark to light, and `DARK_THRESHOLD` for transitioning from light to dark. When the light level crosses the `DARK_THRESHOLD` (higher value), the system switches to the dark state, but it doesn't switch back to light until the light level falls below the `LIGHT_THRESHOLD` (lower value). This creates a dead zone between the two thresholds where no change in state is registered, thus providing a buffer against light level variations and ensuring that the sensor's event changes are deliberate and stable, reducing the likelihood of oscillation or noise-induced changes.

Part of the debugging process was changing the file names to fit our project. Once we made sure all the correct files and functions were renamed we could continue with everything else. When implementing the first version of the bumper event checker with bit shifting, the design was straightforward but the hard part was using bitwise vs logical operators. The struggle was mostly because we have not used them in a while and using them incorrectly results in the wrong operation. For the light sensor event, the design was exactly as the pseudo code but we did not consider that when declaring the curEvent as an undefined value, it could bypass the

hysteresis check and spam the system with no events. This was solved by setting curEvent to the lastEvent so it would always be defined.

Part 7: Finite State Machine

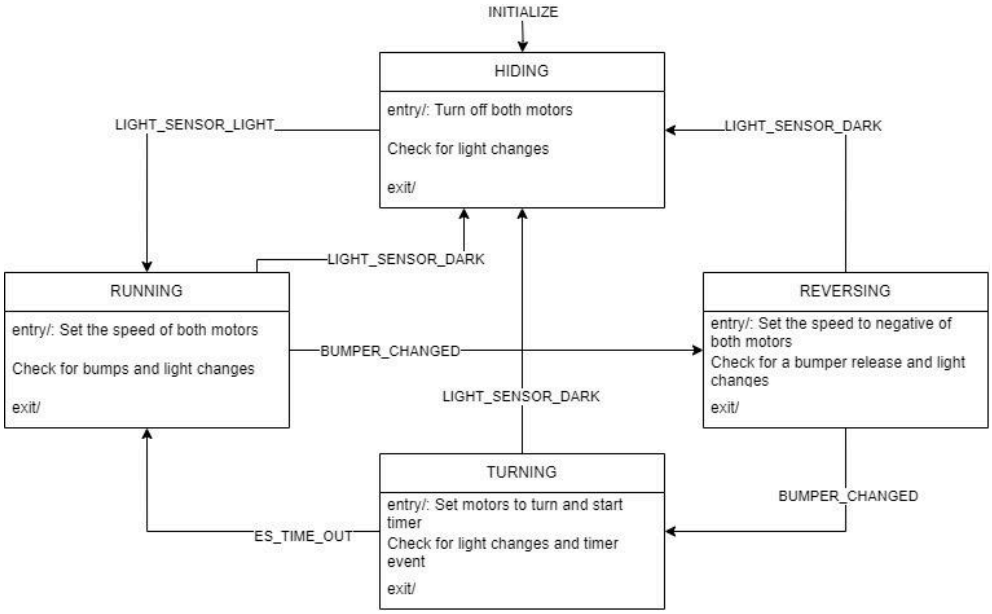

Figure 2. Finite State Machine

From the original state machine, we made a few modifications to include one more state which resulted in four states. We started in keyboard input mode and with the provided state machine and made sure to understand how it worked before implementing our own. Initially we started with adding our own states and transitioning between them using one event to verify that the state machine would move between states. After making sure that the state machine would transition correctly we added the desired motion for the roach. This involved setting the motors and making them turn and reverse. We observed this first with the keyboard and afterwards with the roach on a block. Once we started testing on the ground, we realized some parts needed improvement. After reversing, a second bumper event, indicating a release in the bumper, which

would trigger a halt of reversal, was not triggering a switch in states. This was solved by checking if the parameter of a bump was from a release or a press. This was the only big issue we encountered as the rest were just changing the thresholds to work in certain lighting.

The following is the relevant code required for the functioning for the finite state machine. This code outlines the states and state transitions that the above state diagram defines.

```
ES_Event RunFSM(ES_Event ThisEvent) {

    . . . other code

    switch (CurrentState) {
        case InitPState: // If current state is initial Pseudo State
             . . . other code
            break;

        case Hiding: // in the first state, replace this with appropriate
state
            if (ThisEvent.EventType == ES_ENTRY) {
                Roach_LeftMtrSpeed(0);
                Roach_RightMtrSpeed(0);
                ThisEvent.EventType = ES_NO_EVENT;
            }

            else if (ThisEvent.EventType == LIGHT_SENSOR_LIGHT) {
                nextState = Running;
                makeTransition = TRUE;
                ThisEvent.EventType = ES_NO_EVENT;
            }

            else if (ThisEvent.EventType == ES_EXIT) {
                ThisEvent.EventType = ES_NO_EVENT;
            }
            break;

        case Running:
            if (ThisEvent.EventType == ES_ENTRY) {
                Roach_LeftMtrSpeed(ROACH_SPEED);
                Roach_RightMtrSpeed(ROACH_SPEED);
                ThisEvent.EventType = ES_NO_EVENT;
            }

            else if (ThisEvent.EventType == LIGHT_SENSOR_DARK) {
                nextState = Hiding;
                makeTransition = TRUE;
                ThisEvent.EventType = ES_NO_EVENT;
            }

            else if (ThisEvent.EventType == BUMPER_CHANGED) {
                if (ThisEvent.EventParam == 0x0) {
```

```
                break;
            }
            nextState = Reversing;
            makeTransition = TRUE;
            ThisEvent.EventType = ES_NO_EVENT;
        }

        else if (ThisEvent.EventType == ES_EXIT) {
            ThisEvent.EventType = ES_NO_EVENT;
        }
        break;

    case Reversing:
        if (ThisEvent.EventType == ES_ENTRY) {
            Roach_LeftMtrSpeed(-ROACH_SPEED);
            Roach_RightMtrSpeed(-ROACH_SPEED);
            ThisEvent.EventType = ES_NO_EVENT;
        }

        else if (ThisEvent.EventType == LIGHT_SENSOR_DARK) {
            nextState = Hiding;
            makeTransition = TRUE;
            ThisEvent.EventType = ES_NO_EVENT;
        }

        else if (ThisEvent.EventType == BUMPER_CHANGED) {
            if (ThisEvent.EventParam != 0x0) {
                break;
            }
            nextState = Turning;
            makeTransition = TRUE;
            ThisEvent.EventType = ES_NO_EVENT;
        }

        else if (ThisEvent.EventType == ES_EXIT) {
            ThisEvent.EventType = ES_NO_EVENT;
        }
        break;

    case Turning:
        if (ThisEvent.EventType == ES_ENTRY) { // turn counterclockwise
            Roach_LeftMtrSpeed(-ROACH_SPEED);
            Roach_RightMtrSpeed(ROACH_SPEED);
            ES_Timer_InitTimer(ROACH_TIMER, TURNING_TICKS);
            ThisEvent.EventType = ES_NO_EVENT;
        }

        else if (ThisEvent.EventType == LIGHT_SENSOR_DARK) {
            nextState = Hiding;
            makeTransition = TRUE;
            ThisEvent.EventType = ES_NO_EVENT;
        }

        else if (ThisEvent.EventType == ES_TIMEOUT) {
            nextState = Running;
            makeTransition = TRUE;
```

```
                ThisEvent.EventType = ES_NO_EVENT;
            }

            else if (ThisEvent.EventType == ES_EXIT) {
                ThisEvent.EventType = ES_NO_EVENT;
            }
            break;

        default: // all unhandled states fall into here
            break;
    } // end switch on Current State
    if (makeTransition == TRUE) {
      . . . other code
    }
    ES_Tail(); // trace call stack end
    return ThisEvent;
}
```

The following is a link of the functionality of our FSM on a roach:

https://youtu.be/qtOb4daB8VE

Part 8: Hierarchical State Machine

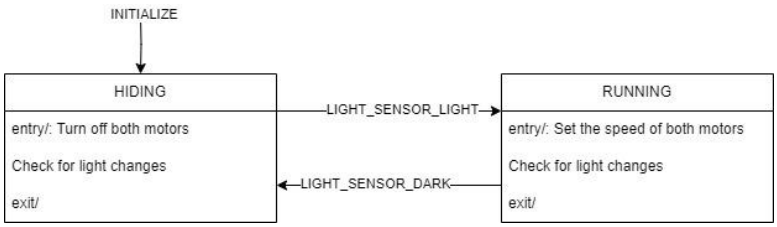

Figure 3. Hierarchical State Machine: Top Level

Designing the top level HSM was relatively straightforward since all the complex behavior would be handled in each state's sub HSMs. For the very basic behaviors of the roach, the two states needed were a "hiding in dark" and "running from light" state. Light Events handled the transitions between the states. Because the complex behavior would be handled in the sub HSMs, these upper level states did not need to implement anything like moving the wheels in specific configurations or listening to bumper events.

The following is the relevant code required for the functioning for the hierarchical state machine.

This code outlines the states and state transitions that the above state diagram defines.

```
ES_Event RunHSM(ES_Event ThisEvent) {
       . . .
     Other Code
     . . .
       switch (CurrentState) {
       . . .
     Other Code
     . . .
       case Hiding: // in the first state, replace this with correct names
              // run sub-state machine for this state
              //NOTE: the SubState Machine runs and responds to events before anything in
the this
              //state machine does
              ThisEvent = RunSubHSMHiding(ThisEvent);
              switch (ThisEvent.EventType) {
              case ES_ENTRY:
                     Roach_LeftMtrSpeed(0);
                     Roach_RightMtrSpeed(0);
                     break;
              case ES_EXIT:
                     // possibly stop timer
                     break;

                     // previous was ES_TIMEOUT
              case LIGHT_SENSOR_LIGHT:
                     nextState = Running;
                     makeTransition = TRUE;
                     ThisEvent.EventType = ES_NO_EVENT;
                     break;

              case ES_NO_EVENT:
                     break;
              default:
                     break;
              }
              break;

       case Running: // running state
              // run sub-state machine for this state
              //NOTE: the SubState Machine runs and responds to events before anything in
the this
              //state machine does
              ThisEvent = RunSubHSMRunning(ThisEvent);
              switch (ThisEvent.EventType) {
              case ES_ENTRY:
                     Roach_LeftMtrSpeed(ROACH_SPEED);
                     Roach_RightMtrSpeed(ROACH_SPEED);
                     break;
              case ES_EXIT:
                     // possibly stop timers
                     break;
              case ES_NO_EVENT:
                     break;
                     // Previous was ES_TIMEOUT
              case LIGHT_SENSOR_DARK:
                     nextState = Hiding;
                     makeTransition = TRUE;
```

```
                        ThisEvent.EventType = ES_NO_EVENT;
                        break;


                default:
                        break;
                }
                break;
. . .
Other Code
. . .

}
```

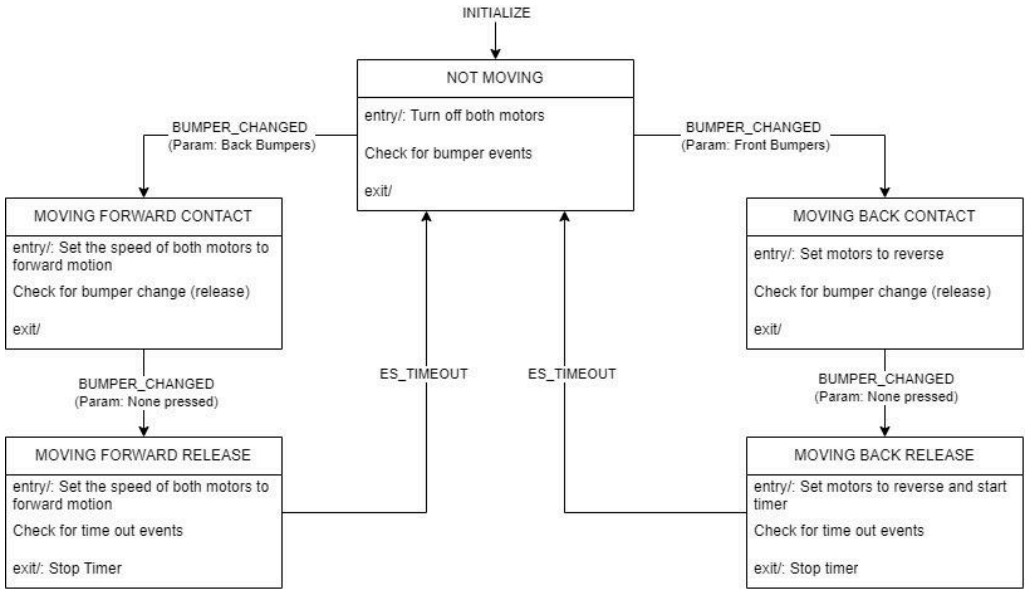

Figure 4. Substate Machine: Hiding

The sub HSM for the upper level “Hiding” state would run whenever the roach was in said upper level state. This sub HSM had to implement the complex behavior associated with “hiding in the dark”, such as avoiding bumps on either side of the roach. Specifically, the roach had to move away from the bumped side until there was no more object pressing down the bumper and then move in that same direction for slightly longer. This was implemented through two additional states for each direction. The first of said states simply set the motors to their appropriate directions and speeds upon being bumped. When no longer bumped, the next state

set a timer to know how much additional time after the bumper release to keep moving for. Once that timeout was processed, the roach went back into stationary to stay in the dark area.

When starting the debugging process, the difficult portion was finding a method to determine correctly which bumpers were pressed. Initially we attempted to have a bitmask but this resulted in some cases being misinterpreted by our system. In the end we used equality comparators to determine which bumpers were pressed. Once we solved that issue it was straight forward as we had tested our state machine many times before we started coding.

The following is a condensed version of the state machine code used for the Hiding substate. The code was written around the given template subHSM files.

```
ES_Event RunSubHSMHiding(ES_Event ThisEvent) {
	uint8_t makeTransition = FALSE; // use to flag transition
	SubHSMState_t nextState; // <- change type to correct enum

	ES_Tattle(); // trace call stack

	switch (CurrentState) {
. . .
other code
. . .
		case NotMoving: // in the first state, replace this with correct names
		  switch (ThisEvent.EventType) {
		  case ES_ENTRY:
			Roach_LeftMtrSpeed(0);
			Roach_RightMtrSpeed(0);
			break;

		  case ES_EXIT:
			break;

		  case BUMPER_CHANGED: //one of the bumpers was pressed, which one?

			if (ThisEvent.EventParam == 0b0001 ||
				ThisEvent.EventParam == 0b0010 ||
				ThisEvent.EventParam == 0b0011) { // one of the front bumpers
was pressed
			nextState = MovingBackContact;
			makeTransition = TRUE;
			ThisEvent.EventType = ES_NO_EVENT;
			} else if (ThisEvent.EventParam == 0b0100 ||
				ThisEvent.EventParam == 0b1000 ||
				ThisEvent.EventParam == 0b1100) { // one of the back bumpers
was pressed
			nextState = MovingForwardContact;
			makeTransition = TRUE;
			ThisEvent.EventType = ES_NO_EVENT;
			} else {
			ThisEvent.EventType = ES_NO_EVENT;
			}
```

```
                        break;

                case ES_NO_EVENT:
                default: // all unhandled events pass the event back up to the next level
                        break;
                }
                break;

        case MovingBackContact: // in the first state, replace this with correct names
                switch (ThisEvent.EventType) {
                case ES_ENTRY:
                        Roach_LeftMtrSpeed(-ROACH_SPEED);
                        Roach_RightMtrSpeed(-ROACH_SPEED);
                        break;

                case ES_EXIT:
                        break;

                case BUMPER_CHANGED: // one of the bumpers was pressed, which one?

                        if (ThisEvent.EventParam == 0) { // one of the front bumpers was
pressed
                        nextState = MovingBackRelease;
                        makeTransition = TRUE;
                        ThisEvent.EventType = ES_NO_EVENT;
                        } else {
                        ThisEvent.EventType = ES_NO_EVENT;
                        }
                        break;

                case ES_NO_EVENT:
                default: // all unhandled events pass the event back up to the next level
                        break;
                }
                break;

        case MovingBackRelease: // in the first state, replace this with correct names
                switch (ThisEvent.EventType) {
                case ES_ENTRY:
                        Roach_LeftMtrSpeed(-ROACH_SPEED);
                        Roach_RightMtrSpeed(-ROACH_SPEED);
                        ES_Timer_InitTimer(ROACH_TIMER, ROACH_TIMERS_TICKS);
                        break;

                case ES_EXIT:
                        ES_Timer_StopTimer(ROACH_TIMER);
                        break;

                case ES_TIMEOUT:
                        nextState = NotMoving;
                        makeTransition = TRUE;
                        ThisEvent.EventType = ES_NO_EVENT;
                        break;

                case ES_NO_EVENT:
                default: // all unhandled events pass the event back up to the next level
                        break;
                }
                break;
        case MovingForwardContact: // in the first state, replace this with correct names
                switch (ThisEvent.EventType) {
                case ES_ENTRY:
                        Roach_LeftMtrSpeed(ROACH_SPEED);
                        Roach_RightMtrSpeed(ROACH_SPEED);
                        break;

                case ES_EXIT:
                        break;

                case BUMPER_CHANGED: // one of the bumpers was pressed, which one?
```

```
                        if (ThisEvent.EventParam == 0) { // one of the front bumpers was
pressed
                        nextState = MovingForwardRelease;
                        makeTransition = TRUE;
                        ThisEvent.EventType = ES_NO_EVENT;
                        } else {
                        ThisEvent.EventType = ES_NO_EVENT;
                        }
                        break;

                case ES_NO_EVENT:
                default: // all unhandled events pass the event back up to the next level
                        break;
                }
                break;
        case MovingForwardRelease: // in the first state, replace this with correct names
                switch (ThisEvent.EventType) {
                case ES_ENTRY:
                        Roach_LeftMtrSpeed(ROACH_SPEED);
                        Roach_RightMtrSpeed(ROACH_SPEED);
                        ES_Timer_InitTimer(ROACH_TIMER, ROACH_TIMERS_TICKS);
                        break;

                case ES_EXIT:
                        ES_Timer_StopTimer(ROACH_TIMER);
                        break;

                case ES_TIMEOUT:
                        nextState = NotMoving;
                        makeTransition = TRUE;
                        ThisEvent.EventType = ES_NO_EVENT;
                        break;

                case ES_NO_EVENT:
                default: // all unhandled events pass the event back up to the next level
                        break;
                }
                break;

        default: // all unhandled states fall into here
                break;
        } // end switch on Current State
        . . .
    Other Code
    . . .
}
```

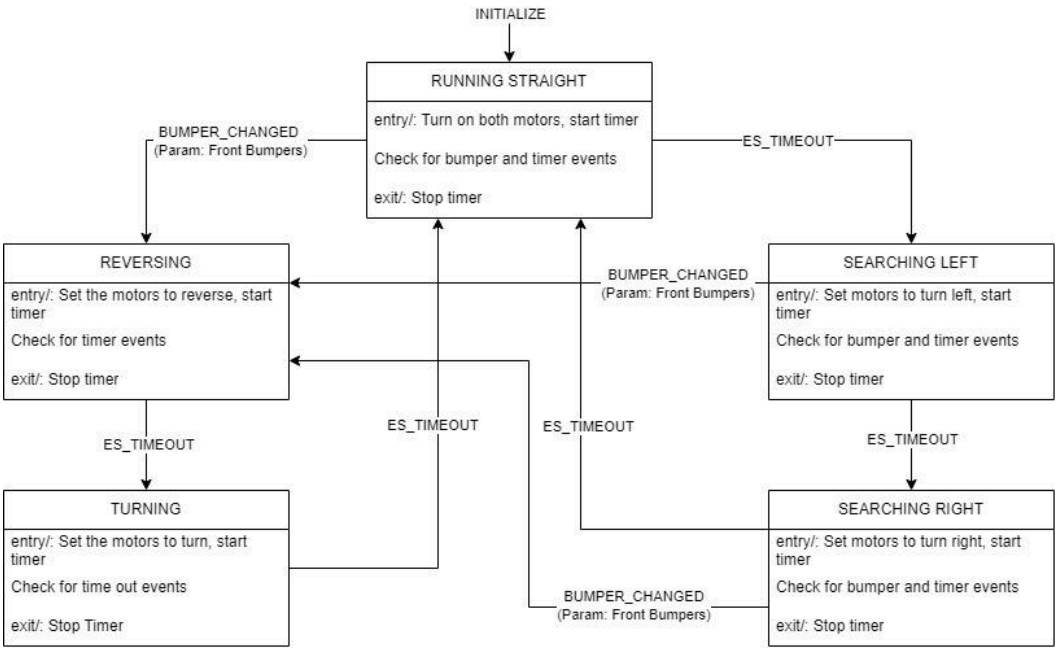

Figure 4. Substate Machine: Running

The sub HSM for the Running State can be observed in Fig. 4 which runs when the upper level HSM is in the Running state and has detected light. It begins in a straightforward movement, monitoring for obstructions via bumper inputs or for pre-set timeouts, which would indicate the need to alter course. On bumper trigger, the machine reverses, then pivots to evade obstacles. When a timeout occurs, it seeks new paths by turning or systematically scanning left and right creating an infinity sign which was our implementation of the dance/search. Having it scan rather than spin would increase its chances of finding darkness. If at any point it received the event for darkness it would exit and pass the event to the top level HSM.

We mapped out all possible paths and made sure the transitions were what we expected using the keyboard inputs. Each one worked as expected since we tested our state machine very well before starting to code. Once we connected all the parts and moved from keyboard inputs to receiving roach inputs it worked well the first time around. After running the code on the roach,

we realized that we wanted to improve upon our search from making and S shape to a complete infinity shape. We also wanted our turns to be 60 degrees rather than 180. Both of these changes only required changing the ticks used for the timer in the different scenarios and then we were finished with the finetuning.

The following is a condensed version of the state machine code used for the Running substate. The code was written around the given template subHSM files.

```
ES_Event RunSubHSMRunning(ES_Event ThisEvent) {
    . . . other code

    switch (CurrentState) {
        case InitPSubState: // If current state is initial Pseudo State
            . . . other code

        case RunningStraight: // in the first state, replace this with correct names
            switch (ThisEvent.EventType) {
                case ES_ENTRY:
                    Roach_LeftMtrSpeed(ROACH_SPEED);
                    Roach_RightMtrSpeed(ROACH_SPEED);
                    ES_Timer_InitTimer(ROACH_TIMER, RUN_TIMER_TICKS);
                    break;

                case ES_EXIT:
                    ES_Timer_StopTimer(ROACH_TIMER);
                    break;

                case ES_TIMEOUT:
                    nextState = SearchingLeft;
                    makeTransition = TRUE;
                    ThisEvent.EventType = ES_NO_EVENT;
                    break;

                case BUMPER_CHANGED:
                    if (ThisEvent.EventParam == 0b0001 ||
                            ThisEvent.EventParam == 0b0010 ||
                            ThisEvent.EventParam == 0b0011) { // one of the front bumpers
was pressed
                        nextState = Reversing;
                        makeTransition = TRUE;
                        ThisEvent.EventType = ES_NO_EVENT;
                    } else {
                        ThisEvent.EventType = ES_NO_EVENT;
                    }

                    break;
                case ES_NO_EVENT:
                default: // all unhandled events pass the event back up to the next level
                    break;
            }
            break;

        case SearchingLeft: // in the first state, replace this with correct names
            switch (ThisEvent.EventType) {
                case ES_ENTRY:
                    Roach_LeftMtrSpeed(ROACH_SPEED/2);
```

```
                Roach_RightMtrSpeed(ROACH_SPEED);
                ES_Timer_InitTimer(ROACH_TIMER, ROACH_DANCING_TICKS);
                break;

            case ES_EXIT:
                ES_Timer_StopTimer(ROACH_TIMER);
                break;

            case BUMPER_CHANGED:
                if (ThisEvent.EventParam == 0b0001 ||
                        ThisEvent.EventParam == 0b0010 ||
                        ThisEvent.EventParam == 0b0011) { // a front bumpers  pressed
                    nextState = Reversing;
                    makeTransition = TRUE;
                    ThisEvent.EventType = ES_NO_EVENT;
                } else {
                    ThisEvent.EventType = ES_NO_EVENT;
                }
                break;

            case ES_TIMEOUT:
                nextState = SearchingRight;
                makeTransition = TRUE;
                ThisEvent.EventType = ES_NO_EVENT;
                break;


            case ES_NO_EVENT:
            default: // all unhandled events pass the event back up to the next level
                break;
        }
        break;
    case SearchingRight: // in the first state, replace this with correct names
        switch (ThisEvent.EventType) {
            case ES_ENTRY:
                Roach_LeftMtrSpeed(ROACH_SPEED);
                Roach_RightMtrSpeed(ROACH_SPEED/2);
                ES_Timer_InitTimer(ROACH_TIMER, ROACH_DANCING_TICKS);
                break;

            case ES_EXIT:
                ES_Timer_StopTimer(ROACH_TIMER);
                break;

            case BUMPER_CHANGED:
                if (ThisEvent.EventParam == 0b0001 ||
                        ThisEvent.EventParam == 0b0010 ||
                        ThisEvent.EventParam == 0b0011) { // a front bumpers was pressed
                    nextState = Reversing;
                    makeTransition = TRUE;
                    ThisEvent.EventType = ES_NO_EVENT;
                } else {
                    ThisEvent.EventType = ES_NO_EVENT;
                }
                break;

            case ES_TIMEOUT:
                nextState = RunningStraight;
                makeTransition = TRUE;
                ThisEvent.EventType = ES_NO_EVENT;
                break;


            case ES_NO_EVENT:
            default: // all unhandled events pass the event back up to the next level
                break;
        }
        break;
    case Reversing: // in the first state, replace this with correct names
        switch (ThisEvent.EventType) {
```

```
                case ES_ENTRY:
                    Roach_LeftMtrSpeed(-ROACH_SPEED);
                    Roach_RightMtrSpeed(-ROACH_SPEED);
                    ES_Timer_InitTimer(ROACH_TIMER, ROACH_AVOID_TICKS);
                    break;

                case ES_EXIT:
                    ES_Timer_StopTimer(ROACH_TIMER);
                    break;

                case ES_TIMEOUT:
                    nextState = Turning;
                    makeTransition = TRUE;
                    ThisEvent.EventType = ES_NO_EVENT;
                    break;

                case ES_NO_EVENT:
                default: // all unhandled events pass the event back up to the next level
                    break;
            }
            break;
        case Turning: // in the first state, replace this with correct names
            switch (ThisEvent.EventType) {
                case ES_ENTRY:
                    Roach_LeftMtrSpeed(-ROACH_SPEED);
                    Roach_RightMtrSpeed(ROACH_SPEED);
                    ES_Timer_InitTimer(ROACH_TIMER, ROACH_TURN_TICKS);
                    break;

                case ES_EXIT:
                    ES_Timer_StopTimer(ROACH_TIMER);
                    break;

                case ES_TIMEOUT:
                    nextState = RunningStraight;
                    makeTransition = TRUE;
                    ThisEvent.EventType = ES_NO_EVENT;
                    break;

                case ES_NO_EVENT:
                default: // all unhandled events pass the event back up to the next level
                    break;
            }
            break;

        default: // all unhandled states fall into here
            break;
    } // end switch on Current State

    . . . other code

    return ThisEvent;
}
```

The following is a link of the functionality of our HSM on a roach:

https://youtu.be/aQDWnOwneKg

Appendix and Notes

Gitlab repository for our code: https://git.ucsc.edu/adiazroq/ece118_lab0