Last modification: February 14, 2020

RBE 1001: Introduction to Robotics
C-Term 2019-20
HW 5.1: State machines and event-driven programming

This is an individual assignment. Each team member must complete this before starting on HW 5.2.

When programming a microcontroller to perform a set of tasks, it is often useful to organize the system as a *state machine*. By assigning states to the system, programming is made much more efficient (and readable!), which speeds up development time and makes the system more reliable. In this exercise, you'll build a nominal "home alarm system," and you will be shown how to program the system as a state machine. In the process, you will also learn about *events* and using events to control your system's behavior.

Objectives

Upon successful completion of this lab, the student will be able to:

- Implement checker-handler construction for events,
- Implement a state machine, and
- Demonstrate their knowledge by building a home alarm system.

Events

A common function for any system is to react to a particular input with a specific response. In previous tutorials, you pushed a button and an LED turned on or off. Such input-response is ubiquitous to automated system. But how the system registers and reacts to an input can greatly affect the system's behavior, as we'll demonstrate with an example.

Imagine you run a ball factory. You want to count how many balls roll out of each ball-making machine, so you set up a sensor that adds to a counter whenever it detects a ball passing in front of it. You might be tempted to write code that looks something like the following:

```
while the machine is running
  if a ball is present
    add one to your counter
end
```

Will this work? Why not?

The answer lies in the fact that the counter is triggered by the *presence* of a ball. If the `while` loop runs reasonably fast, then every time a ball rolls by, it will get counted *many* times and lead to a gross overestimation. A better way would be to count the *arrival* of balls:

```
while the machine is running
  if a ball is present now and was not present before
    add one to your counter
end
```

This logic will ensure that each ball is counted as it arrives at the sensor, and the counter cannot be further incremented until the ball leaves and another one arrives, guaranteeing an accurate count. In other words, the counter is triggered by the *event* of the ball arriving. An event occurs whenever there is a *change* in some condition: a button is pressed, a laser beam is broken, a timer expires.

Coding for events

To properly detect events, it is necessary to declare a variable that holds the previous value of an input and compare it to the latest value. Here's an example for our ball detector in `C++`:

```
bool CheckForNewBall(void)
{
  static bool prevReading = false;

  bool retVal = false;
  bool currReading = ReadSensor(); //reads true if there's a ball; otherwise false
  if(prevReading == false && currReading == true)
  {
    retVal = true;
  }

  prevReading = currReading;
  return retVal;
}
```

The key is the line that looks to see if the previous reading was `false` and the current reading is `true`. Of the four possible combinations of the two variables, this is the only one that corresponds to the arrival of a ball. How would you change it if you wanted to capture the departure?

Some other rules-of-thumb to note:

- You should only read the sensor once and then use that value for the remainder of the routine. The reason is that if you read the sensor multiple times in the same routine, there is a small, but finite, chance that the value will change between readings.
- You must update the value of `prevReading` before you return from the routine.
- Though it requires a little more code, best practice is to carry a return value, in this case `retVal`, through the entire function, updating it as needed and returning it at the end.
- Checkers (and the associated handlers presented below) should all run very fast. You don't want to miss an important event because you're busy in another routine. The `delay()` function, while sometimes useful, should be used very sparingly.

- The `static` declaration gives `prevReading` global *persistence* (but local *scope*). You should be familiar with these terms from your CS classes.

When an event occurs, it needs to be handled. As such, another good practice is to structure code with *event-handler* pairs. Using our ball detector example, event-handler construction might look something like:

```
void loop()
{
    if(CheckBallDetector()) HandleDetectionEvent();
}
```

where the handler in this case is the nearly trivial,

```
void HandleDetectionEvent(void)
{
  ballCount++;
}
```

The code could also be made cleaner by encapsulating everything into a class.

If you want more information on Arduino syntax, see the Appendix and the tutorials therein.

State machines

As noted in the introduction, it is often useful to think of an automated system as a *state machine*. A state machine consists of a set of independent states and a set of rules for how the system changes from one state to another. At any point in time the state machine can be in only one of the possible states, and that state describes the system completely. The system moves between the states in response to events, which trigger one or more *actions*. Figure 1 shows a pair of generic states, a transition between them, and the nomenclature we use for the diagrams.

To give a concrete example, consider an alarm clock. Figure 2 shows a state machine for a typical alarm clock with a snooze option. Starting in the `OFF` state, let's walk through the diagram. When the user sets an alarm, the system moves from `OFF` to `RUNNING`. While in the `RUNNING` state, the system checks to see of the clock time has passed the alarm time. When it does, the system starts buzzing and changes to the `ALARMING` state. While in the `ALARMING` state, if the user presses the snooze button, it does two things: turns off the buzzer and starts a snooze timer. When in the `SNOOZING` state, if the snooze timer expires, the system turns the buzzer back on and goes back to the `ALARMING` state. In any state, the user can cancel the alarm, which takes the system back to the `OFF` state.

Coding a state machine

We noted above that the system will transition from one state to another when a particular event occurs, for example the system goes from `ALARMING` to `SNOOZING` when the snooze button is pressed.

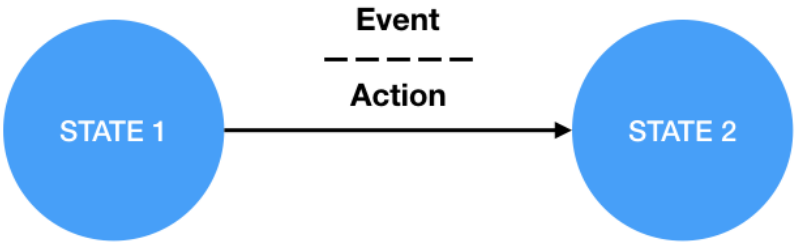

Figure 1: A pair of generic states and a transition between them.

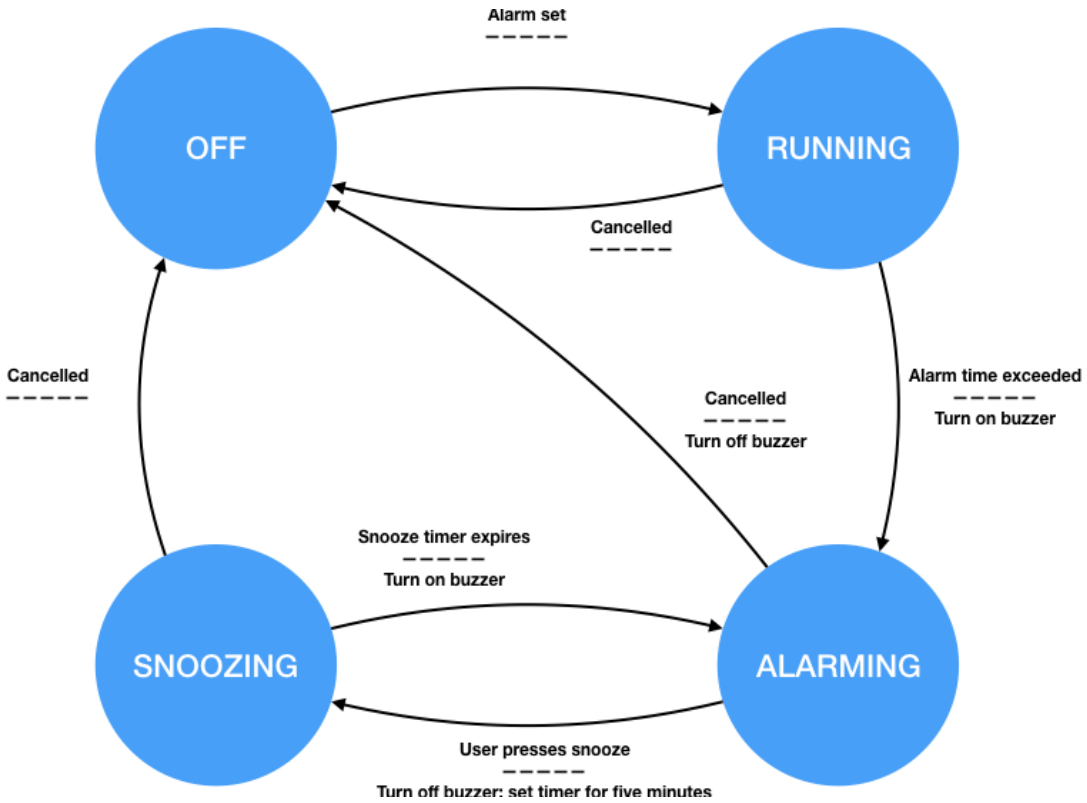

Figure 2: State machine for an alarm clock.

Note, however, that not all events have meaning for every state. For example, in our alarm example, pressing the snooze button has no effect when the system is in the `OFF` state.

Practically, what this means is that the system only needs to respond to a given event when it's in a state where that event is meaningful. The easiest way to manage this in code is to use a *state variable* to track which state your system is in. To make the code more readable, you can use the `enum` construction, which assigns numerical values to text descriptors, as follows:

```
enum {OFF, RUNNING, ALARMING, SNOOZING};
int currentState = OFF;
```

Here we declare a set of states with four possible values. We then declare a specific variable, `currentState`, which will be used to keep track of system's current state. From there, it's a matter of coding the state machine through conditional statements. For the event of pressing the snooze button, this might look something like the following in pseudo-code:

```
begin HandleSnoozeButton
  if current state is ALARMING
    turn off the buzzer
    start a timer for five minutes
    change current state to SNOOZING
  else if in any other state
    ignore
  end if
end
```

Note that pressing the snooze button only has meaning when in the `ALARMING` state – in other states it is ignored. For practical reasons, it is often useful to use the `switch` construct for actual code, but the syntax is a little complicated, so we won't spend time on that here (here's a tutorial, if you're interested). Just keep in mind that it's a convenient way to organize a bunch of `if..else` statements when you get to that point in your project.

Name _____

State transition diagrams

This is an individual assignment!

1. In your next assignment, you will be building a simplified home alarm system, but first, you'll need to plan out the system using a state diagram. Your alarm system will have the following functionality:

   - To arm the system, the user will push a button,
   - When the system is armed, an LED will shine on a photoresistor and a servo motor will move roughly 135 degrees (to signify locking a door),
   - If an object comes between the LED and the photoresistor while the system is armed, an alarm will sound,
   - To disarm the system, regardless of whether or not it is alarming, the user will press a second button,
   - When disarmed, the LED will turn off and the servo motor will move back to the "unlocked" poistion. If the system is alarming, the sound will turn off, as well.

   Draw out the state diagram for your alarm system. Be sure to include the states, events, and actions using the proper format shown in Figure 1.

2. Using pseudo-code (or `C++`), write out a checker function for testing if the laser beam is broken. You *must* use proper event checking!

3. Explain what the `static` declaration for a variable does. Why doesn't `prevReading` get reset to `false` every time `CheckBallDetector()` is called?