RBE 2002: Unified Robotics II
Lab 2 Addendum: State machines

Last modification: October 29, 2022

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.

Events

A common function for any system is to react to a particular input with a specific response. 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, ensuring 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 often 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 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 checker routine. The `delay()` function is generally a bad decision.
- The `static` declaration gives `prevReading` global *persistence*, but local *scope*. You should be familiar with these terms from your CS classes. Better is to encapsulate everything in a class (**but don't use** `static` **then** – that has a different meaning for 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 trivial,

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

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

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. Within a state, the system might be doing something (e.g., line following), and 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.

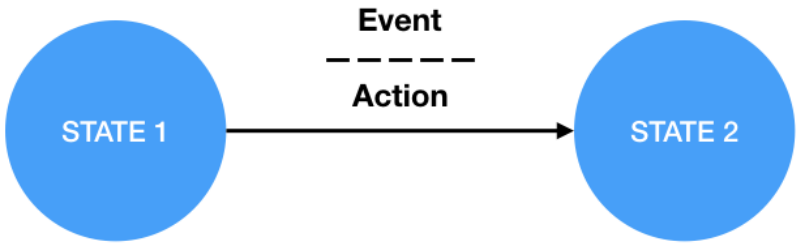

Figure 1: A pair of generic states and a transition between them.

To give a concrete example, consider an alarm clock. Figure 2 shows one possible state machine for an 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 takes two actions: 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.

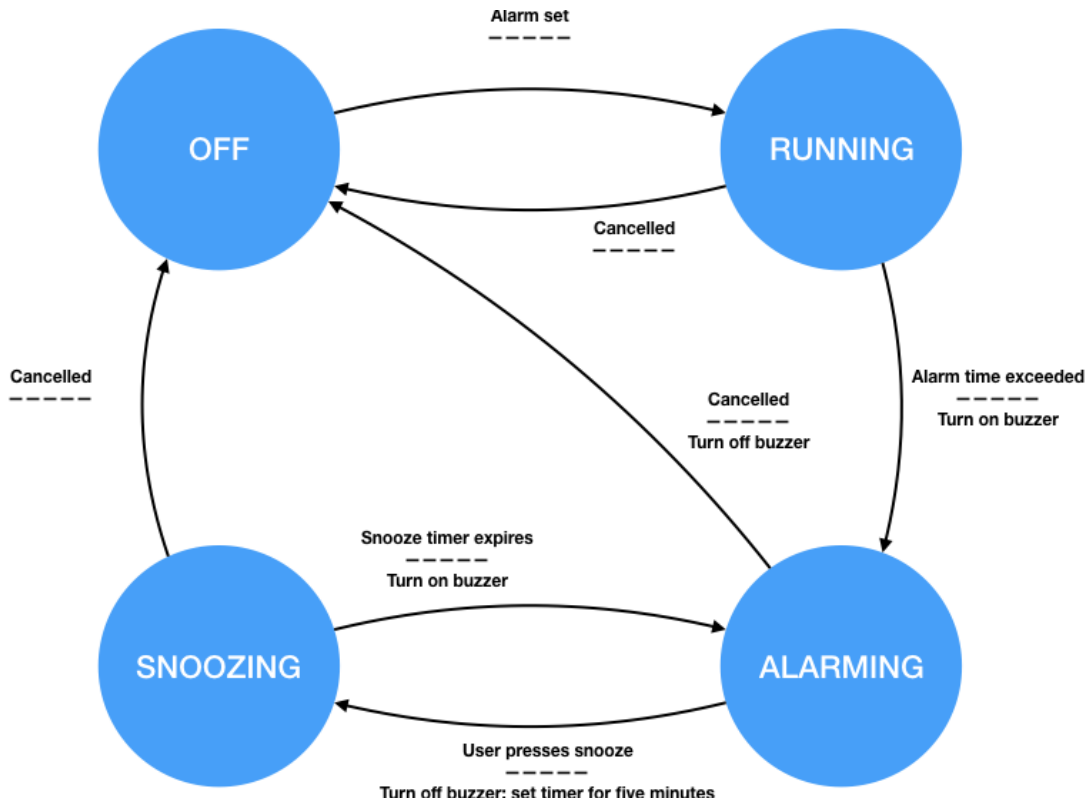

Figure 2: State machine for an alarm clock.

Note, however, that not all events have meaning for every state. 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, which is a convenient way to organize a bunch of `if..else` statements. Here is a resource, if you're interested).