**University of California, Santa Cruz**

**COSMOS Cluster 6:**

**Networking and Robotics**

**Lab 3: Events and Services**

## Introduction

In the previous lab, you implemented a choreographed sequence of actions on the RoachBot. The timing was implemented with blocking code. This worked well enough as long as the bot did not need to perform other activities at the same time, but as soon as an LED display was added to the spec, the system broke down.

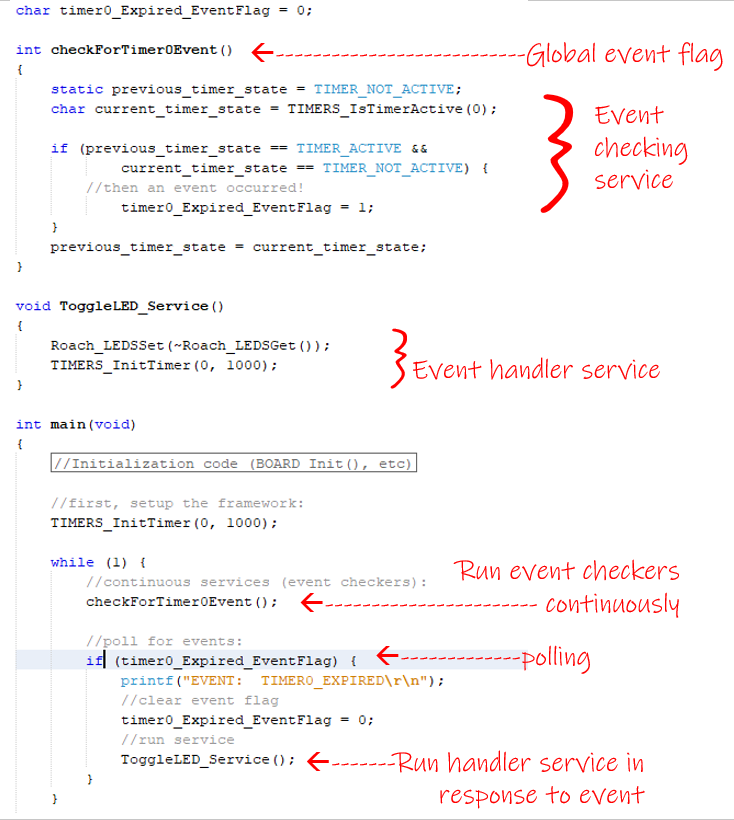
In this lab, we’ll learn how to write code in an *asynchronous* way. Synchronous code, like the type we’ve been writing, executes sequentially. The code may loop, or branch on if/else statements, or take detours through function calls, but it still goes from top top bottom

Asynchronous code, by contrast, is *reactive.* The code is written in “services”, ( also called “handlers” or “callbacks”) which execute in reaction to events. This allows us to write code for many different behaviors that can execute more or less at the same time.

## The Events-and-Services Paradigm:

In many languages (for example, Java) the asynchronous behavior is built-in. Java classes often have built-in methods like “onButtonPress()” which are run when a user presses a button. But C doesn’t work this way! It’s purely synchronous. So we need a framework for building an asynchronous system using synchronous code.[[1]](#footnote-1)

Here’s an example:



Hopefully that’s not too overwhelming! Let’s break down the parts of this:

1. A global variable that contains events. In this case, it’s just a Boolean (sometimes called an “event flag”.
2. A service that is mean runs continuously. Its only job is to detect events and throw them. These types of services are often called “event checkers.”
3. An event handling service that runs in response to a particular event.
   1. Note that this service restarts the timer that triggered it, ensuring it will run again. This is how you make a service run periodically.
4. A main loop that manages the events-and-services system. It:
   1. Runs event checking services
   2. Polls for events
      1. If it detects an event, it runs the event handler services and clears (or “consumes”) the event.

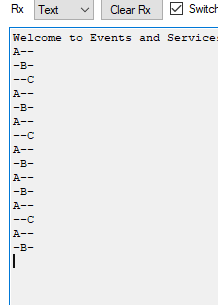
This framework is at the heart of every reactive coding environment. It’s how the Java VM works, and it’s in your microwave, your watch, and every embedded device anywhere (well, every *well-coded* embedded device anyway).

Now let’s see if we can use it:

#### Instructions:

1. Run the above code and verify that it behaves appropriately.
2. Modify the above code so that it has 3 events and 3 services.
   1. One service prints the string “A--\r\n” every 2 seconds.
   2. One service prints the string “-B-\r\n” every 3 seconds.
   3. One service prints the string “--C\r\n” every 5 seconds.

It should look a bit like this (though, depending on how you set up the timers, it may look a little different). You can also find Timers\_example.hex in the lab zipfile, if you want to try it yourself.



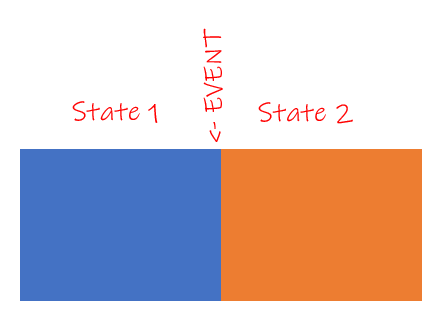
OPTIONAL CHALLENGES:

In addition to printing, can you make the 2-second service toggle the red LEDs, the 3-second service toggle the yellow LEDs, and the 5-second service toggle the green LEDs?

The above code involves a lot of writing! Can you think of a way to write it with only one event checking service, and only one global event variable? (of course, that variable can’t be a Boolean)

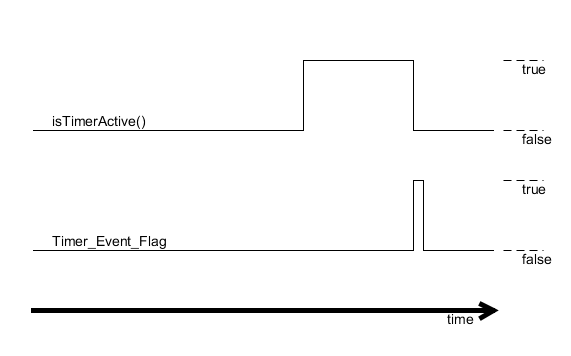
## Event-checking in more detail

It’s worth looking a little deeper into event checking. Events are supposed to be *instantaneous,* representing a change between two states[[2]](#footnote-2). They should have a very brief existence. Here’s a visual analogy:



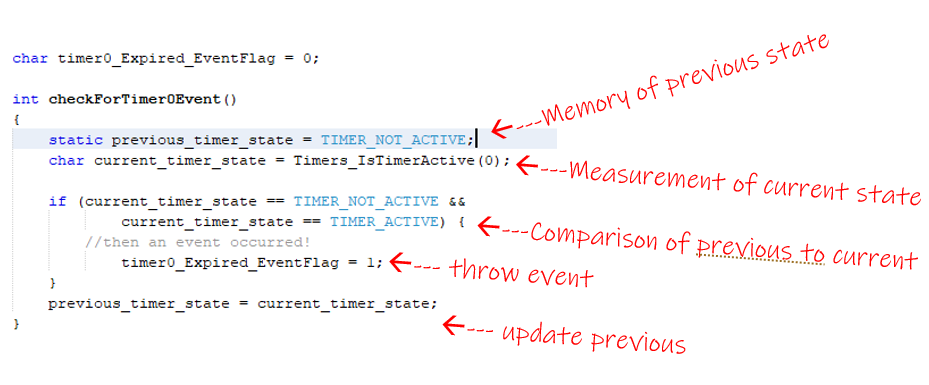
The blue and orange rectangles are states, and they occupy area. The *event* is the boundary between them, and it occupies no area.

This is analogous to our timer event checker:



The event is the boundary between the duration when the timer is active and when it is not active. The event is very nearly instantaneous[[3]](#footnote-3)

So how do we implement this in code? Let’s take a closer look at our timer event checker:



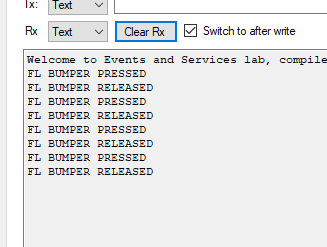
There are 5 essential components to any event checker:

1. A record of the previous state.
   1. Many novices try to write event checkers without this. It’s not going to work! It is fundamentally impossible to detect a change between two measurements without recording the previous measurement!
2. A new measurement of the current state.
   1. This is where you read your sensors. Ideally, it is the *only* place in your code where you read your sensors.
3. A comparison between the previous measurement and the current one.
4. Throwing (or generating) the event,
5. Updating the previous record

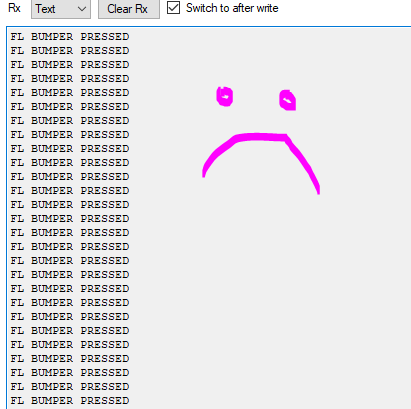
#### Instructions:

One type of event that is very important for programming the roach is a bump event. If the front left bumper is suddenly pressed, our system should detect a “FL\_bumper\_pressed” event. When that bumper is released, our system should detect a bump “FL\_bumper\_released” event.

1. Implement this event checker. When you use it, it should look like this:



1. It should NOT look like this (this is called “spamming”, where events are continuously generated, and it will absolutely mess up everything real quck):



CHALLENGES:  
 Add an event checker for the light sensor. It should throw two kinds of events, LIGHT\_TO\_DARK and DARK\_TO\_LIGHT. You’ll need to figure out a suitable threshold between the two.

Extra Challenge:

Implement the challenge from Lab2: Make the roach navigate the maze, while displaying a cycling LED pattern.

1. You might be wondering *why* C doesn’t work this way. The answer is that computer processors are synchronous, under the hood. In order to get ansynchronous behavior, you have write an asynchronous system on top of a synchronous system – and C is built to do this! In fact,, the Java Virtual Machine, which runs Java Code, is written in C. [↑](#footnote-ref-1)
2. In the sense of “state of mind” or “state of distress,” not “state of California.” [↑](#footnote-ref-2)
3. in practice, it isn’t quite instantaneous, because instructions take time to execute! But ideally, we can limit its duration to the time it takes for one iteration of the while loop. [↑](#footnote-ref-3)