Lab 4: State machines

UCR EE/CS 120B

Embedded systems commonly have time-ordered behavior (more so than in desktop systems). C was not originally intended for time-ordered behavior. Trying to code time-ordered behavior directly with C's sequential statement computation model results in countless variations of "spaghetti" code. Instead, a disciplined programming approach captures behavior using a state machine computation model.

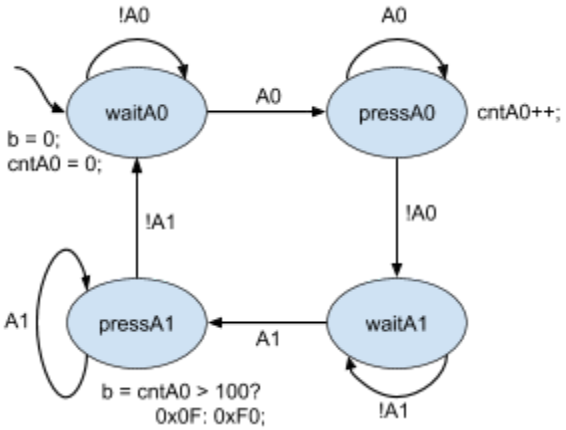
Implement all code from here on out in C using the *standard techniques* (Ch. 3.4) found in "Programming Embedded Systems," PES (Vahid/Givargis/Miller).

Pre-lab

Read through the lab manual and (1) draw a state machine for exercise 1 and 2 by hand and (2) write a `tests.gdb` file for both exercise 1 and 2 targeting the ATmega1284.

Testing State Machines

When testing state machines we want to input a sequence of values and inspect the final output and/or final state to verify correctness. Additionally, we want to have more fine grained control over the *starting state* of the program. We will go over an example of both using the below state machine and this implementation.



State Sequence

Let's write a test to see if we can go from '`waitA0`' to '`pressA1`'

```
# Test sequence from waitA0: A0, !A0, A1 => PORTB: 1
test "PINA: 0x01, 0x00, 0x02 => PORTB: 1, state: pressA1"
set state = waitA0
setPINA 0x01
continue 2
setPINA 0x00
continue 2
setPINA 0x02
continue 2
expectPORTB 0xF0
checkResult
```

Notice how we continue several times in between each change to `PINA`. When you program the hardware next week, the clock is ticking so fast that each button press will last for several runs through the program, we want to test to make sure our state machines work in those situations.

Since '`pressA1`' has an output, we are able to inspect `PORTB` to see if we're in the right state, but we should inspect the `state` variable as well. We can add this line under `expectPORTB 0xF0` to check that the state is in fact in '`pressA1`'.

```
expect state pressA1
```

Starting State

Each test we write should:

1. Initialize the context to some known starting state
   `(gdb) set state = waitA0`
2. Apply stimulus (input)
   `(gdb) setPINA 0x01`
   `...`
3. Observe output
   `(gdb) expectPORTB 0xF0`
   `(gdb) expect state pressA1`

Let's write a test to see if we can get `PORTB` output to be `0x0F` when our `cntA0` is greater than 100.

```
test "cntA0 > 100 => PORTB: 0x0F"
set exampleTick::cntA0 = 101
set state = pressA1
setPINA 0x02
continue 2
expectPORTB 0x0F
expect state pressA1
checkResult
```

Let's look at a couple specific lines in this test:

- "`set exampleTick::cntA0 = 101`" will set the `cntA0` variable that is locally scoped in the `exampleTick()` function to the value of 101. Whenever you are setting or inspecting a variable that is local to a function (except main) you must scope it.
- "`set state = pressA1`" will set the current state to `pressA1`
- "`setPINA 0x02`" sets A1 so we will stay in state `pressA1`

We then continue several times and verify that the state doesn't change and the outpt on `PORTB` is correct (`0x0F`).

Exercises

Note: Drawing state machines is a helpful and powerful way to debug your own code. If you are running into a problem in your logic, drawing out the SM will help both you, and your TA, to analyze your logic.

Write C programs for the following exercises targeting an ATmega1284 following the PES *standard technique*. For any behavior response caused by a button press, the response should occur almost immediately upon the press, not waiting for the button release (unless otherwise stated). Be sure to count each button press only once, no matter the duration the button is pressed. In addition to demoing your programs, you should show that your code adheres entirely to the technique in PES for capturing SMs in C, with no variations.

Note: As you are using the standard model, you should `display` the `state` variable to ensure you are transitioning properly between states. When demoing be sure to have this ready for the TA.

Note: When developing, you want to save your work every so often so that you don't lose your work if your machine dies (or, in the case of the lab machines, you accidentally log out or get disconnected). You especially want to save when you are about to implement a new feature so if you break anything you can *revert* back to an older, working, copy. **This will be the last lab we remind you to do this!**

If feasible, demonstrate parts 1 and 2 to a TA before moving on.

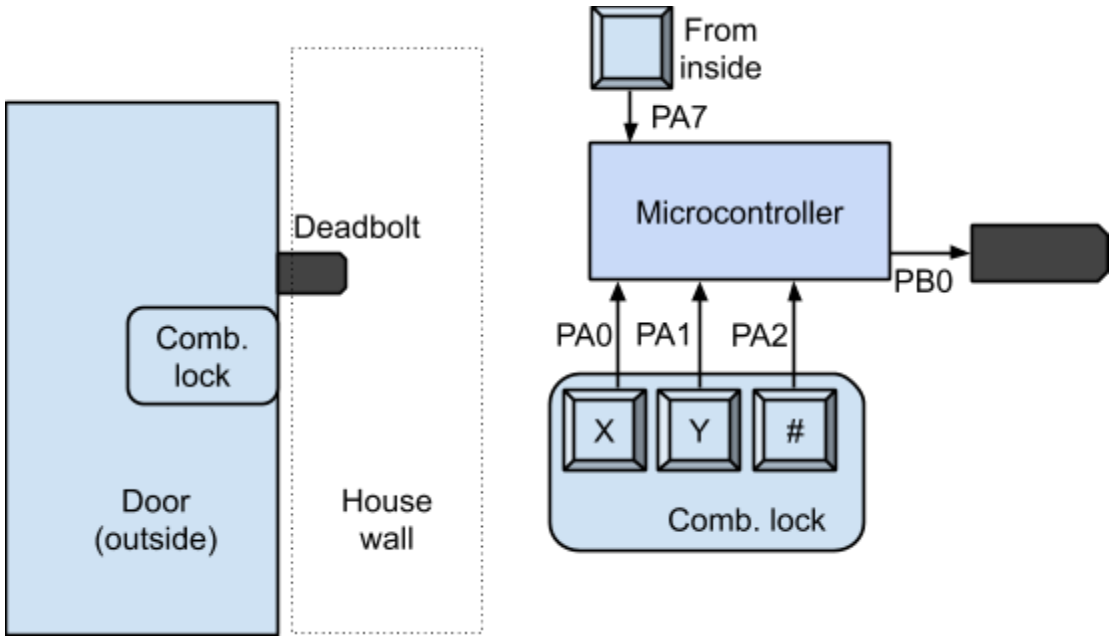
1. `PB0` and `PB1` each connect to an LED, and `PB0`'s LED is initially on. Pressing a button connected to `PA0` turns off `PB0`'s LED and turns on `PB1`'s LED, staying that way after button release. Pressing the button again turns off `PB1`'s LED and turns on `PB0`'s LED.

   ```
   $ cp source/main.c turnin/[cslogin]_lab4_part1.c
   $ cp test/tests.gdb turnin/[cslogin]_lab4_part1_tests.gdb
   $ git commit -m "Completed part 1"
   ```

2. Buttons are connected to `PA0` and `PA1`. Output for `PORTC` is initially **7**. Pressing `PA0` increments `PORTC` once (stopping at 9). Pressing `PA1` decrements `PORTC` once (stopping at 0). If both buttons are depressed (even if not initially simultaneously), `PORTC` resets to 0.

   ```
   $ cp source/main.c turnin/[cslogin]_lab4_part2.c
   $ cp test/tests.gdb turnin/[cslogin]_lab4_part3_tests.gdb
   $ git commit -m "Completed part 2"
   ```

3. A household has a digital combination deadbolt lock system on the doorway. The system has buttons on a keypad. Button 'X' connects to PA0, 'Y' to PA1, and '#' to PA2. Pressing and releasing **'#'**, then pressing **'Y'**, should unlock the door by setting PB0 to 1. Any other sequence fails to unlock. Pressing a button from inside the house (PA7) locks the door (PB0=0). For debugging purposes, give each state a number, and always write the current state to PORTC (consider using the enum state variable). Also, be sure to check that only one button is pressed at a time.




```
$ cp source/main.c turnin/[cslogin]_lab4_part3.c
$ cp test/tests.gdb turnin/[cslogin]_lab4_part3_tests.gdb
$ git commit -m "Completed part 3"
```

4. (**Challenge**) Extend the above door so that it can also be *locked* by entering the earlier code.
   ```
   $ cp source/main.c turnin/[cslogin]_lab4_part4.c
   $ cp test/tests.gdb turnin/[cslogin]_lab4_part4_tests.gdb
   $ git commit -m "Completed part 4"
   ```
5. (**Challenge**) Extend the above door to require the 4-button sequence **#-X-Y-X** rather than the earlier 2-button sequence. To avoid excessive states, store the correct button sequence in an array, and use a looping SM.
   ```
   $ cp source/main.c turnin/[cslogin]_lab4_part5.c
   $ cp test/tests.gdb turnin/[cslogin]_lab4_part5_tests.gdb
   $ git commit -m "Completed part 5"
   ```

Submission

Each student must submit their source files (`.c`) and test files (`.gdb`) according to instructions in the lab submission guidelines.

```
$ tar -czvf [cslogin]_lab2.tgz turnin/
```

Don't forget to commit and push to Github before you logout!