UCR EE/CS 120B

Zyante Chapter 3: State Machines

Lab 3: State machines (Digital lock)

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).**

Note: When developing code, get in the habit of making backup copies of your source file after every 10 minutes or so of work, so you can revert to earlier (especially working) versions.

Pre-lab

Draw a state machine for exercise 1 by hand or in RIBS and then implement that state machine by writing C code for the ATmega1284. **Do not use RIBS to auto-generate the C code** -- type the C by hand, following the *standard technique* in PES §3.4 "Implementing an SM in C".

Note: RIBS will use the PES standard technique to generate C code from a finite state machine (FSM). While we want you to write the pre-lab by hand, RIBS/RIMS can give a good example of how to write code using the standard technique from an SM.

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.

Atmel Studio 7 optimizes the template SM code's enum that is used for the state variable. Here's how to tell Atmel Studio 7 to stop optimizing enum:

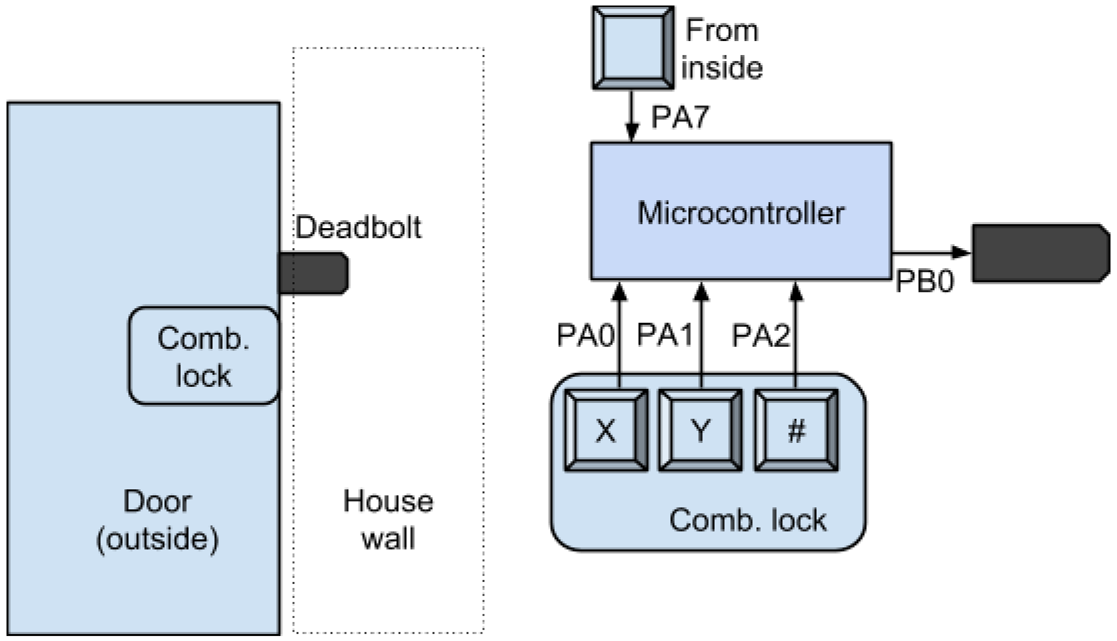
- Right-click the project in the solution explorer, click on "Toolchain", then under AVR/GNU C Compiler click "Optimization".
- Then, change Optimization Level to "None (-O0)" and uncheck "Allocate only as many bytes needed by enum types (-fshort-enums).

Write C programs for the following exercises using Atmel Studio for 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 add the state variable to the watch list to ensure you are transitioning properly between states. When demoing be sure to have this ready for the TA.

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.
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.
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.



4. (**Challenge**) Extend the above door so that it can also be *locked* by entering the earlier code.
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.

One student from each group must submit the group's .c source files according to instructions in the lab submission guidelines. Post any questions or problems you encounter to the wiki and discussion boards on iLearn.

Don't forget to commit and push your code to Github before shutting down the VM!