EE/CS 120B: Introduction to Embedded Systems
University of California, Riverside
Winter 2016

Laboratory Exercise 3

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 for this laboratory exercise in C by adopting the templates found in Sections 3.4 (for SMs) and 4.7 (for synchSMs) of the PES zyBook. The adaptations should replace the RIMS API with an interface for the ATmega1284.

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 versions.

Part 1 – SMs Design

Pre-lab

NOTE: Please also complete Part2's Pre Lab on page 7.
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. Type the C code by hand as described in the "Programming Embedded Systems" zyBook; do not use RIBS.

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.

AVR Studio 6 optimizes the template SM code's enum that is used for the state variable. Here's how to tell AVR Studio 6 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 AVR Studio for an ATMega1284. Set up the breadboard and implement each exercise using buttons, LEDs, etc. 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. To add a variable to watch, first enter debug mode. Then, right click the variable you wish to add and Select "Add to Watch".

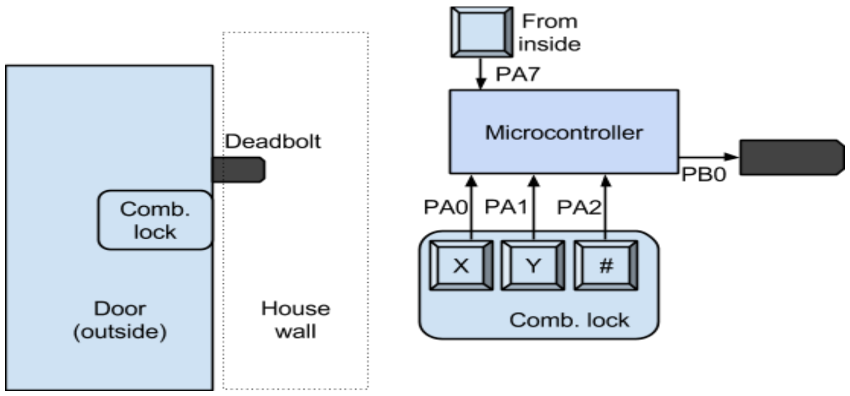
If feasible, demonstrate parts 1 and 2 to a TA before moving on.

1. PB0 and PB1 each connect to an LED. PB0's LED is initially on, PB1 is initially off. 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 binary value 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). Be sure to check that only one button is pressed at a time.



Challenge Problems

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

5. 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.

Each 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.

Part 2 – SynchSM Design

Like most microcontrollers, the ATmega1284 has built-in timers. Also like most microcontrollers, the ATmega1284's timers have numerous low-level configurable options. However, our disciplined embedded programming approach, as described in PES, uses a clean abstraction of a timer involving just a few simple functions:

- void TimerSet(unsigned char M) -- set the timer to tick every M milliseconds
- void TimerOn() -- inititialize and start the timer
- void TimerOff() -- stop the timer
- void TimerISR() -- called automatically when the timer ticks, with contents filled by the user ONLY with an instruction that sets the user-declared global variable TimerFlag=1

The following program contains variable and function declarations that map the ATmega1284's low-level timer constructs to the above clean abstraction. **Do NOT copy and paste the code. Type it out by hand to avoid errors.**

```
#include <avr/io.h>
#include <avr/interrupt.h>

// TimerISR() sets this to 1. C programmer should clear to 0.
volatile unsigned char TimerFlag = 0;

// Internal variables for mapping AVR's ISR to our cleaner TimerISR model.
unsigned long _avr_timer_M = 1; // Start count from here, down to 0. Default 1 ms.
unsigned long _avr_timer_cntcurr = 0; // Current internal count of 1ms ticks

void TimerOn() {

        // AVR timer/counter controller register TCCR1
        // bit3 = 0: CTC mode (clear timer on compare)
        // bit2bit1bit0=011: pre-scaler /64
        // 00001011: 0x0B
        // SO, 8 MHz clock or 8,000,000 /64 = 125,000 ticks/s
        // Thus, TCNT1 register will count at 125,000 ticks/s
        TCCR1B = 0x0B;

        // AVR output compare register OCR1A.
        // Timer interrupt will be generated when TCNT1==OCR1A
        // We want a 1 ms tick. 0.001 s * 125,000 ticks/s = 125
        // So when TCNT1 register equals 125,
        // 1 ms has passed. Thus, we compare to 125.
        OCR1A = 125;
```

```
	// AVR timer interrupt mask register
	// bit1: OCIE1A -- enables compare match interrupt
	TIMSK1 = 0x02;

	//Initialize avr counter
	TCNT1=0;

	// TimerISR will be called every _avr_timer_cntcurr milliseconds
	_avr_timer_cntcurr = _avr_timer_M;

	//Enable global interrupts:  0x80: 1000000
	SREG |= 0x80;
}

void TimerOff() {
	// bit3bit1bit0=000: timer off
	TCCR1B = 0x00;
}

void TimerISR() {
	TimerFlag = 1;
}

// In our approach, the C programmer does not touch this ISR, but rather TimerISR()
ISR(TIMER1_COMPA_vect) {

	// CPU automatically calls when TCNT1 == OCR1
	// (every 1 ms per TimerOn settings)

	// Count down to 0 rather than up to TOP (results in a more efficient comparison)
	_avr_timer_cntcurr--;
	if (_avr_timer_cntcurr == 0) {

		// Call the ISR that the user uses
		TimerISR();
		_avr_timer_cntcurr = _avr_timer_M;
	}
}

// Set TimerISR() to tick every M ms
void TimerSet(unsigned long M) {
	_avr_timer_M = M;
	_avr_timer_cntcurr = _avr_timer_M;
}
```

```
void main()
{
	// Set port B to output
	// Init port B to 0s
	DDRB = 0xFF;
	PORTB = 0x00;

	TimerSet(1000);
	TimerOn();
	unsigned char tmpB = 0x00;
	while(1) {
		// User code (i.e. synchSM calls)

		// Toggle PORTB; Temporary, bad programming style
		tmpB = ~tmpB;
		PORTB = tmpB;

		// Wait 1 sec
		while (!TimerFlag);
		TimerFlag = 0;
		// Note: For the above a better style would use a synchSM with TickSM()
		// This example just illustrates the use of the ISR and flag
	}
}
```

The above sample main code toggles port B every 1 second (**Note**: main's code is used for simple illustration and is itself not good style). Load the project into the chip memory. Any LEDs connected to port B pins should now blink on 1 sec and off 1 sec. (If you're curious, you can try setting the oscillator fuse to a 1 MHz setting, and you'll see slower blinking).

Pre-lab

Draw your synchSM for exercise 1, and write out the C code. Do not use RIBS to auto-generate the C code.

Exercise

Implement the following. For each, create a synchSM, then implement in C, and map to your board. You may optionally use RIBS to help generate code for exercises other than exercise 1.

1. Create a synchSM to blink three LEDs connected to PB0, PB1, and PB2 in sequence, 1 second each. Implement that synchSM in C as described in Section 4.7 of the 'Programming Embedded Systems' zyBook. In addition to demoing your program, you will need to show that your code adheres entirely to the method with no variations.

   Video Demonstration: **http://youtu.be/ZS1Op26WiBM**

Challenge Problem

2. Create a simple light game that requires pressing a button on PA0 while the middle of three LEDs on PB0, PB1, and PB2 is lit. The LEDs light for 300 ms each in sequence. When the button is pressed, the currently lit LED stays lit. Pressing the button again restarts the game.

   Video Demonstration: **http://youtu.be/inmzsXz_HG0**

Each student must submit their .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.