Lab 6: SynchSMs

UCR EE/CS120B

Pre-lab

Read the entire lab manual and draw your synchSM for exercise 1, and write out the `tests.gdb` file for exercise 1 targeting the ATmega1284.

Timer abstraction

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.** It might be helpful to have the code below in a header file (i.e. timer.h) (not including `#include <avr/io.h>, #ifdef` to `#endif (lines 3-5)` and the main function since those belong in main.c). If you do make a timer.h, make sure to include `#include <avr/interrupt.h>` to avoid errors in later labs.

```
#include <avr/io.h>
#include <avr/interrupt.h>
#ifdef _SIMULATE_
#include "simAVRHeader.h"
#endif

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

```c
void TimerOn() {
	// AVR timer/counter controller register TCCR1
	TCCR1B = 0x0B;// 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

	// AVR output compare register OCR1A.
	OCR1A = 125;  // 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.
	// AVR timer interrupt mask register
	TIMSK1 = 0x02; // bit1: OCIE1A -- enables compare match interrupt

	//Initialize avr counter
	TCNT1=0;

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

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

// The “enable global interrupts” line is SREG |= 0x80, not SREG |- 0x80;

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

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)
	_avr_timer_cntcurr--; // Count down to 0 rather than up to TOP
	if (_avr_timer_cntcurr == 0) { // results in a more efficient compare
		TimerISR(); // Call the ISR that the user uses
		_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;
}
```

```c
void main() {
        DDRB = 0xFF; // Set port B to output
        PORTB = 0x00; // Init port B to 0s
        TimerSet(1000);
        TimerOn();
        unsigned char tmpB = 0x00;
        while(1) {
                // User code (i.e. synchSM calls)
                tmpB = ~tmpB; // Toggle PORTB; Temporary, bad programming style
                PORTB = tmpB;
                while (!TimerFlag); // Wait 1 sec
                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.

Note: If you port B is blinking every 10-ish seconds or so, type and enter `make fuses` to fix it.

Exercises

Implement the following. For each, create a synchSM, then implement in C, and map to your board.

1. Create a synchSM to blink three LEDs connected to `PB0`, `PB1`, and `PB2` in sequence, 1 second each. Implement that synchSM in C using the method defined in class. In addition to demoing your program, you will need to show that your code adheres entirely to the method with no variations.

   To clarify and match the video, the lights should turn on in the following sequence:

   `PB0,PB1,PB2,PB0,PB1,PB2,PB0,PB1,PB2…`

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

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.

   To clarify and match the video, the lights should turn on in the following sequence:

   `PB0,PB1,PB2,PB1,PB0,PB1,PB2,PB1,PB0…`

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

3. (**Challenge**) (from an earlier lab) Buttons are connected to `PA0` and `PA1`. Output for `PORTB` is initially **7**. Pressing `PA0` increments `PORTB` once (stopping at 9). Pressing `PA1` decrements `PORTB` once (stopping at 0). If both buttons are depressed (even if not initially simultaneously), `PORTB` resets to 0. ***Now that we have timing****, only check to see if a button has been pressed every 100 ms. Additionally, if a button is held, then the count should continue to increment (or decrement) at a rate of once per second.*
   Video Demonstration: http://youtu.be/D33pn3TcjpM

Submission

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

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