Lab 11: Task Scheduler
(2 days)

UCR EE/CS120B

Pre-lab

Read the entire lab manual and have your board wired up and ready to use. Complete the `GetKeyPad()` function and be able to demo its fully working functionality (0 ~ 9, A ~ D, *, #). Be able to demo your LCD still works.

Introduction

In this lab we will introduce a structure for designing a task scheduler for state machines. We will build a simple task scheduler that will process state machines according to the period specified by each state machine task. We will use the scheduler to implement a producer-consumer problem where we take input via a keypad, and use the LCD to output the characters pressed on the keypad.

#include

As we add more functionality to our code it can become a bit cluttered. You are welcome to continue copy and pasting all of the support code directly into the .c file. Alternatively, you may use the following .h files to increase readability.

If you choose to download and use these .h files, a tutorial on how to include these files can be found here

Keypad

A keypad is comprised of several buttons. If each button had its own pin, the keypad below would require 16 pins.

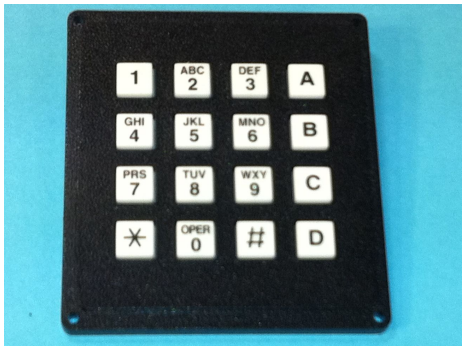


Figure 1: Keypad GH5004-ND

To reduce pin count, keypads commonly have a row/column arrangement as shown below.

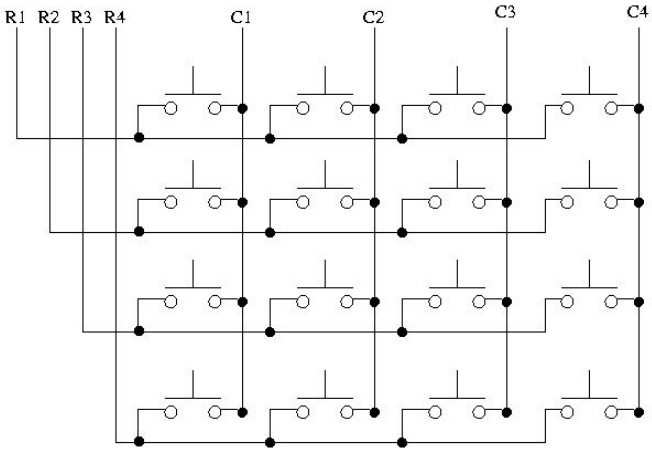


Figure 2: High-Level Connection diagram for Keypad

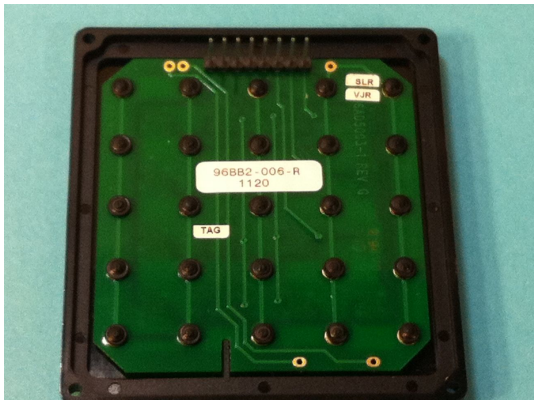


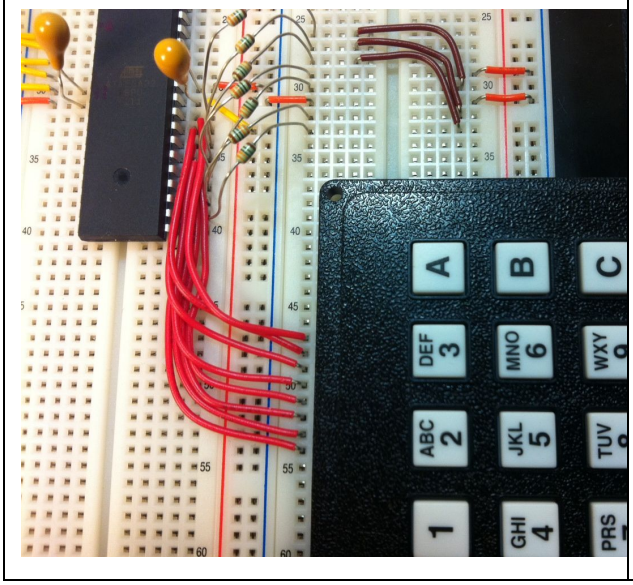
Figure 3: Keypad GH5004-ND Pins - C4C3C2C1 R4R3R2R1 (left to right in figure)

Each row has a pin (R1-R4), and each column has a pin (C1-C4), for a total of 8 pins. Pressing a button uniquely connects one column pin with one row pin. For example, pressing the upper-left button connects pin C1 with pin R1. Pressing the bottom-right button connects C4 and R4. Datasheet

To accomplish accepting input from 16 buttons with only 8 pins a technique known as time multiplexing is employed. The idea is simple, we shall use common row wires and common column wires the achieve our lower pin count. This however causes a problem, by sharing the rows and columns we have cross talk. To overcome this, we will selectively enable one column at a time, check the 4 pins connected to that row, and then continue by enabling the next column, repeating the process for all columns. This is **time multiplexing** -- simultaneous transmission of several messages along a single channel of communication by having those signals transmit at specific times (in this case a specific sequence).

We can get away with this because the microcontroller can operate much faster than humans can react/perceive. In the time it takes a person to press one of the buttons (10's ms), the microcontroller (8,000,000 clock ticks/sec) can make *many* passes of the keypad to check for input. Thus the process is transparent to the user.

Connect the keypad to port C as shown (R1 connected to PC0, …, C4 to PC7).



Actually, you can move the keypad further left for more stability (partly covering the wires).

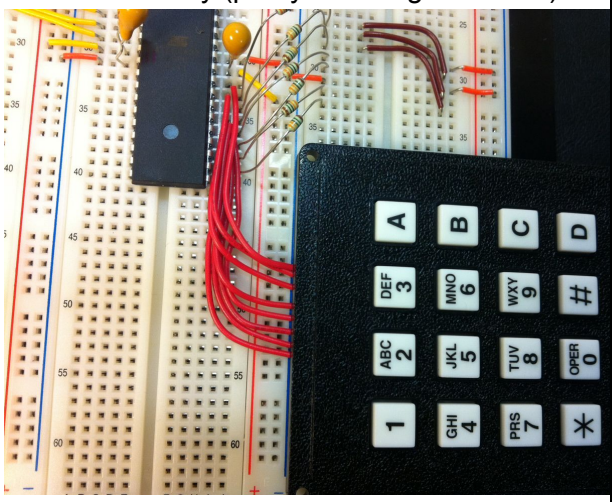


Figure 4: Shown setup

Keypad Connections

Keypad Pin #	1	2	3	4	5	6	7	8
Term	R1	R2	R3	R4	C1	C2	C3	C4
AVR Port	C0 Output	C1 Output	C2 Output	C3 Output	C4 Input	C5 Input	C6 Input	C7 Input

In order to get a correct keypad input, each **term** C1-C4 from figure 2 must be checked if the voltage is logical low; the code belows shows the checking of each column.

The following keypad test program repeatedly scans the keypad and checks for particular buttons being pressed, lighting five LEDs on port B accordingly. The program is unfinished but should work for buttons 1, 2, and *. Put five LEDs on PB4-PB0 and test the program.

***Note**: Don't forget to uncheck the JTAG fuse as we are using port C.*

```
// Returns '\0' if no key pressed, else returns char '1', '2', ... '9', 'A', ...
// If multiple keys pressed, returns leftmost-topmost one
// Keypad must be connected to port C
/* Keypad arrangement
        PC4 PC5 PC6 PC7
   col  1   2   3   4
row
PC0 1   1 | 2 | 3 | A
PC1 2   4 | 5 | 6 | B
PC2 3   7 | 8 | 9 | C
PC3 4   * | 0 | # | D
*/
unsigned char GetKeypadKey() {
       PORTC = 0xEF; // Enable col 4 with 0, disable others with 1's
       asm("nop"); // add a delay to allow PORTC to stabilize before checking
       if (GetBit(PINC,0)==0) { return('1'); }
       if (GetBit(PINC,1)==0) { return('4'); }
       if (GetBit(PINC,2)==0) { return('7'); }
       if (GetBit(PINC,3)==0) { return('*'); }

       // Check keys in col 2
       PORTC = 0xDF; // Enable col 5 with 0, disable others with 1's
       asm("nop"); // add a delay to allow PORTC to stabilize before checking
       if (GetBit(PINC,0)==0) { return('2'); }
       // ... *****FINISH*****

       // Check keys in col 3
       PORTC = 0xBF; // Enable col 6 with 0, disable others with 1's
       asm("nop"); // add a delay to allow PORTC to stabilize before checking
       // ... *****FINISH*****

       // Check keys in col 4
       // ... *****FINISH*****

       return('\0'); // default value
}
```

```c
int main(void)
{
	unsigned char x;
	DDRB = 0xFF; PORTB = 0x00; // PORTB set to output, outputs init 0s
	DDRC = 0xF0; PORTC = 0x0F; // PC7..4 outputs init 0s, PC3..0 inputs init 1s
	while(1) {
		x = GetKeypadKey();
		switch (x) {
			case '\0': PORTB = 0x1F; break; // All 5 LEDs on
			case '1': PORTB = 0x01; break; // hex equivalent
			case '2': PORTB = 0x02; break;

			// . . . ***** FINISH *****

			case 'D': PORTB = 0x0D; break;
			case '*': PORTB = 0x0E; break;
			case '0': PORTB = 0x00; break;
			case '#': PORTB = 0x0F; break;
			default: PORTB = 0x1B; break; // Should never occur. Middle LED off.
		}
	}
}
```

Building the scheduler

A scheduler is code whose purpose is, given multiple tasks, to execute each task at the appropriate time. PES describes a task scheduler in detail. First, we encapsulate all of the information that represents a process in a `task struct`.

```c
//--------Task scheduler data structure---------------------------------------
// Struct for Tasks represent a running process in our simple real-time operating system.
typedef struct _task {
	/*Tasks should have members that include: state, period,
		a measurement of elapsed time, and a function pointer.*/
	signed char state; //Task's current state
	unsigned long int period; //Task period
	unsigned long int elapsedTime; //Time elapsed since last task tick
	int (*TickFct)(int); //Task tick function
} task;

//--------End Task scheduler data structure-----------------------------------
```

The heart of each `task` is the function that it will be executing (`int (*TickFct)(int)`). Each of these functions will be defined as a global function and we use function pointers in the `task struct` to point to the appropriate function. Function pointers work just like a pointer to a `char` or `int`, but they have some specific syntax on how they must be called and defined.
Note: One additional change is we now pass the state variable for each task as part of the function call; there is no longer a global state variable.

For more information on function pointers see: http://www.newty.de/fpt/fpt.html

Now we need to define some state machines to test our scheduler with. We will design a simple system that has a pause button that will toggle with each press of A0 (pauseButtonSM). When the system is paused the output is static, when it is not paused, the LEDs on B0 (toggledLED0SM) and B1 (toggledLED1SM) will toggle on and off. The following shared global variables will be used to communicated between the SMs:

```
//--------Shared Variables----------------------------------------------------
unsigned char led0_output = 0x00;
unsigned char led1_output = 0x00;
unsigned char pause = 0;
//--------End Shared Variables------------------------------------------------
```

The pauseButtonSM will set the `pause` variable:

```
//Enumeration of states.
enum pauseButtonSM_States { pauseButton_wait, pauseButton_press, pauseButton_release };

// Monitors button connected to PA0.
// When button is pressed, shared variable "pause" is toggled.
int pauseButtonSMTick(int state) {
	// Local Variables
	unsigned char press = ~PINA & 0x01;

	switch (state) { //State machine transitions
		case pauseButton_wait:
			state = press == 0x01? SpauseButton_press: pauseButton_wait; break;
		case pauseButton_press:
			state = pauseButton_release; break;
		case pauseButton_release:
			state = press == 0x00? pauseButton_wait: pauseButton_press; break;
		default: state = pauseButton_wait; break;
	}
	switch(state) { //State machine actions
		case pauseButton_wait: break;
		case pauseButton_press:
			pause = (pause == 0) ? 1 : 0; // toggle pause
			break;
		case pauseButton_release: break;
	}
	return state;
}
```

Each of the toggleLEDn state machines will toggle an LED based on the variable `pause`.

```
//Enumeration of states.
enum toggleLED0_States { toggleLED0_wait, toggleLED0_blink };

// If paused: Do NOT toggle LED connected to PB0
// If unpaused: toggle LED connected to PB0
int toggleLED0SMTick(int state) {
	switch (state) { //State machine transitions
		case toggleLED0_wait: state = !pause? toggleLED0_blink: toggleLED0_wait; break;
		case toggleLED0_blink: state = pause? toggleLED0_wait: toggleLED0_blink; break;
		default: state = toggleLED0_wait; break;
	}
	switch(state) { //State machine actions
		case toggleLED0_wait: break;
		case toggleLED0_blink:
			led0_output = (led0_output == 0x00) ? 0x01 : 0x00; //toggle LED
			break;
	}
	return state;
}
```

```
//Enumeration of states.
enum toggleLED1_States { toggleLED1_wait, toggleLED1_blink };

// If paused: Do NOT toggle LED connected to PB1
// If unpaused: toggle LED connected to PB1
int toggleLED1SMTick(int state) {
	switch (state) { //State machine transitions
		case toggleLED1_wait: state = !pause? toggleLED1_blink: toggleLED1_wait; break;
		case toggleLED1_blink: state = pause? toggleLED1_wait: toggleLED1_blink; break;
		default: state = toggleLED1_wait; break;
	}
	switch(state) { //State machine actions
		case toggleLED1_wait: break;
		case toggleLED1_blink:
			led1_output = (led1_output == 0x00) ? 0x01 : 0x00; //toggle LED
			break;
	}
	return state;
}
```

And, finally, the display SM will combine the shared variables `led0_output` and `led1_output` and push the outputs to `PORTB`.

```
//Enumeration of states.
enum display_States { display_display };

// Combine blinking LED outputs from toggleLED0 SM and toggleLED1 SM, and output on PORTB
int displaySMTick(int state) {
	// Local Variables
	unsigned char output;

	switch (state) { //State machine transitions
		case display_display: state = display_display; break;
		default: state = display_display; break;
	}
	switch(state) { //State machine actions
		case display_display:
			output = led0_output | led1_output << 1; // write shared outputs
                                                     // to local variables
			break;
	}
	PORTB = output;// Write combined, shared output variables to PORTB
	return state;
}
```

Notice how only the first line of the pauseButtonSM and the last line of the displaySM actually interact with the hardware. The other SMs are logical only, no dependence on the hardware.

Now, we are ready to combine all of our SMs in the `main` function:This

```c
// Implement scheduler code from PES.
int main() {
	DDRA = 0x00; PORTA = 0xFF;
	DDRB = 0xFF; PORTB = 0x00;

	//Declare an array of tasks
	static _task task1, task2, task3, task4;
	_task *tasks[] = { &task1, &task2, &task3, &task4 };
	const unsigned short numTasks = sizeof(tasks)/sizeof(task*);

	// Task 1 (pauseButtonToggleSM)
	task1.state = start;//Task initial state.
	task1.period = 50;//Task Period.
	task1.elapsedTime = task1.period;//Task current elapsed time.
	task1.TickFct = &pauseButtonToggleSMTick;//Function pointer for the tick.
	// Task 2 (toggleLED0SM)
	task2.state = start;//Task initial state.
	task2.period = 500;//Task Period.
	task2.elapsedTime = task2.period;//Task current elapsed time.
	task2.TickFct = &toggleLED0SMTick;//Function pointer for the tick.
	// Task 3 (toggleLED1SM)
	task3.state = start;//Task initial state.
	task3.period = 1000;//Task Period.
	task3.elapsedTime = task3.period; // Task current elasped time.
	task3.TickFct = &toggleLED1SMTick; // Function pointer for the tick.
	// Task 4 (displaySM)
	task4.state = start;//Task initial state.
	task4.period = 10;//Task Period.
	task4.elapsedTime = task4.period; // Task current elasped time.
	task4.TickFct = &SMTick4; // Function pointer for the tick.

	// Set the timer and turn it on
	TimerSet(/*GCD*/);
	TimerOn();

	unsigned short i; // Scheduler for-loop iterator
	while(1) {
		for ( i = 0; i < numTasks; i++ ) { // Scheduler code
			if ( tasks[i]->elapsedTime == tasks[i]->period ) { // Task is ready to tick
				tasks[i]->state = tasks[i]->TickFct(tasks[i]->state); // Set next state
				tasks[i]->elapsedTime = 0; // Reset the elapsed time for next tick.
			}
			tasks[i]->elapsedTime += /*GCD*/;
		}
		while(!TimerFlag);
		TimerFlag = 0;
	}
	return 0; // Error: Program should not exit!
}
```

This code assumes that you will calculate the GCD of all of the state machines each time. We are trying to minimize the points in the code that need to change when an SM is added, therefore minimizing the likelihood of making a simple mistake in future iterations of the code. We can set up a short snippet of code that, with the aid of a `findGCD` helper function, will automatically calculate our system period (GCD).

```
//--------Find GCD function --------------------------------------------------
unsigned long int findGCD(unsigned long int a, unsigned long int b)
{
        unsigned long int c;
        while(1){
                c = a%b;
                if(c==0){return b;}
                a = b;
                b = c;
        }
        return 0;
}
//--------End find GCD function ----------------------------------------------
```

```
unsigned long GCD = tasks[0]->period;
for ( i = 1; i < numTasks; i++ ) {
        GCD = findGCD(GCD,tasks[i]->period);
}
```

Exercises

1. Modify the keypad code to be in an SM task. Then, modify the keypad SM to utilize the simple task scheduler format. All code from here on out should use the task scheduler.
2. Use the LCD code, along with a button and/or time delay to display the message *"CS120B is Legend... wait for it DARY!"* The string will not fit on the display all at once, so you will need to come up with some way to paginate or scroll the text.
   Note: If your LCD is exceptionally dim, adjust the resistance provided by the potentiometer connected to Pin #3.

Video Demonstration: http://youtu.be/eAtBTUr_cm8

3. Combine the functionality of the keypad and LCD so when keypad is pressed and released, the character of the button pressed is displayed on the LCD, and stays displayed until a different button press occurs (May be accomplished with two tasks: LCD interface & modified test harness).

Video Demonstration: http://youtu.be/ZCadEA3ryPM

4. (**Challenge**) Notice that you can visually see the LCD refresh each character (display a lengthy string then update to a different lengthy string). Design a system where a single character is updated in the displayed string rather than the entire string itself. Use the functions provided in “io.c”.

   An example behavior would be to initially display a lengthy string, such as

"Congratulations!". The first keypad button pressed changes the first character 'C' to the button pressed. The second keypad press changes the second character to the second button pressed, etc. No refresh should be observable during the character update.

Video Demonstration: http://youtu.be/M_BC9Vualt8

5. (**Challenge**) Using both rows of the LCD display, design a game where a player controlled character avoids oncoming obstacles. Three buttons are used to operate the game. **Criteria:**
    a. Use the cursor as the player controlled character.
    b. Choose a character like '#', '*', etc. to represent the obstacles.
    c. One button is used to pause/start the game.
    d. Two buttons are used to control the player character. One button moves the player to the top row. The other button moves the player to the bottom row.
    e. A character position change should happen immediately after pressing the button.
    f. Minimum requirement is to have one obstacle on the top row and one obstacle on the bottom row. You may add more if you are feeling up to the challenge.
    g. Choose a reasonable movement speed for the obstacles (100ms or more).
    h. If an obstacle collides with the player, the game is paused, and a "game over" message is displayed. The game is restarted when the pause button is pressed.

Hints:

- Due to the noticeable refresh rate observed when using `LCD_DisplayString`, instead use the combination of `LCD_Cursor` and `LCD_WriteData` to keep noticeable refreshing to a minimum.
- LCD cursor positions range between 1 and 32 (NOT 0 and 31).
- As always, dividing the design into multiple, smaller synchSMs can result in a cleaner, simpler design.

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

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!