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

Laboratory Exercise 7

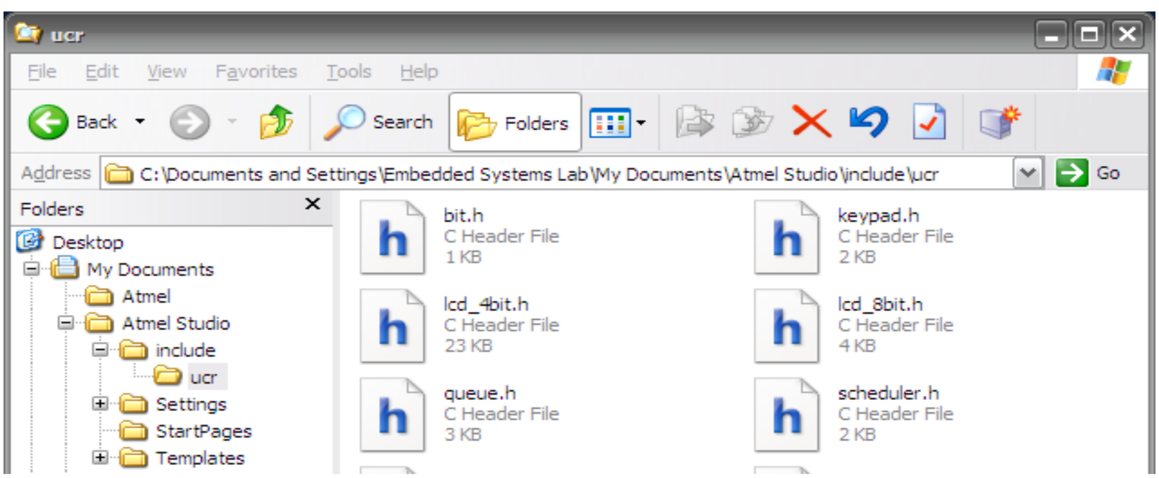
This laboratory exercise introduces a structure for designing a simple task scheduler for concurrent state machines. The task scheduler processes state machines according to the period specified by each state machine task. The scheduler will be used to implement a producer-consumer problem using a keypad for input and an LCD display 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 header files (provided in the .zip file with the Laboratory Exercise) to increase readability. If you choose to use the header files, you may also wish to consult the tutorial on how to include them (also included in the .zip file).

If you are using the CS 120B Project template we have already included the pathway to the following directory shown below. You are welcome to place your .h files here:

C:\Documents and Settings\Embedded Systems Lab\My Documents\Atmel Studio\include\ucr



If you do not use the .h files, you will need to copy paste the appropriate support code into the keypad and LCD code below.
The .h files are designed to work with the task scheduler.

Keypad

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

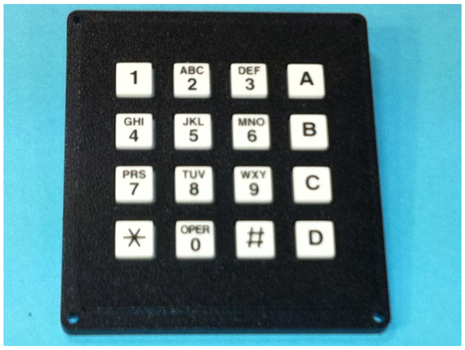


Figure 1: Keypad GH5004-ND

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

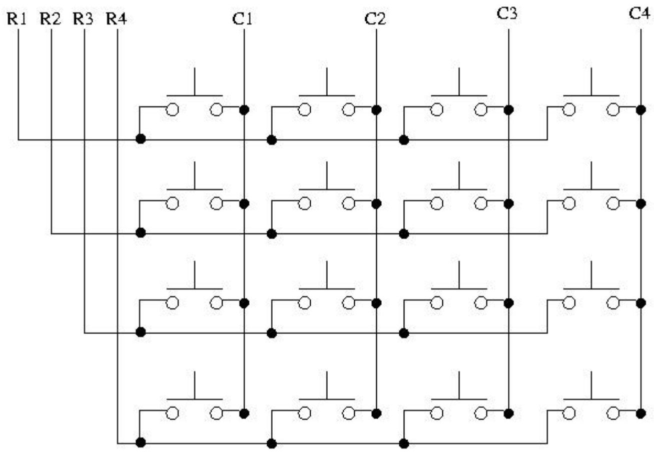


Figure 2: High-Level Connection diagram for Keypad

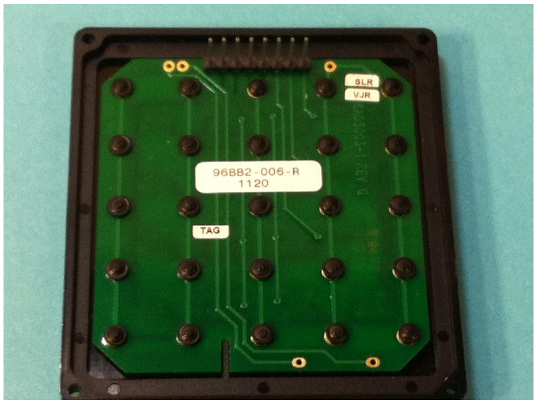
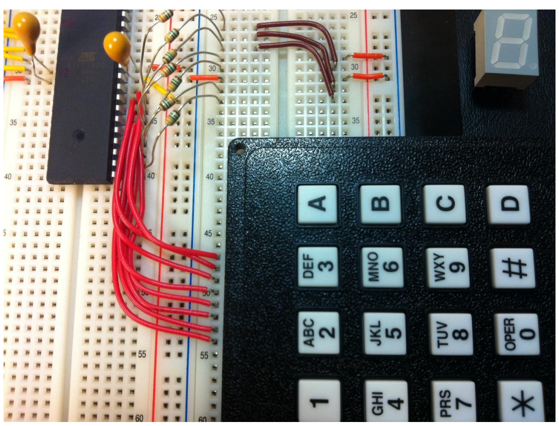


Figure 3: Keypad GH5004-ND Pins - C4C3C2C1 R4R3R2R1 (left to right in figure)

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.

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, the microcontroller 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).



You can move the keypad further left for more stability (partly covering the wires).

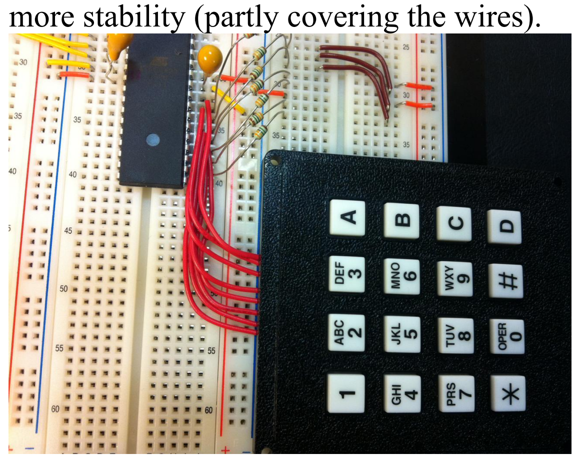


Figure 4: Shown setup

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.*

```c
#include <avr/io.h>
#include <ucr/bit.h>
// 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() {

     // Check keys in col 1
     // Enable col 4 with 0, disable others with 1's
     // The delay allows PORTC to stabilize before checking
     PORTC = 0xEF
     asm("nop");
     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
     // Enable col 5 with 0, disable others with 1's
     // The delay allows PORTC to stabilize before checking
     PORTC = 0xDF;
     asm("nop");
     if (GetBit(PINC,0)==0) { return('2'); }
     // ... *****FINISH*****

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

     // Check keys in col 4
     // ... *****FINISH*****
     return('\0'); // default value
}
```

```c
int main(void)
{
	unsigned char x;

	// PORTB set to output, outputs init 0s
	DDRB = 0xFF; PORTB = 0x00;

	// PC7..4 outputs init 0s, PC3..0 inputs init 1s
	DDRC = 0xF0; PORTC = 0x0F;

	while(1) {
		x = GetKeypadKey();
		switch (x) {

			// All 5 LEDs on
			case '\0': PORTB = 0x1F; break;

			// hex equivalent
			case '1': PORTB = 0x01; break;
			case '2': PORTB = 0x02; break;

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

			case 'D': PORTB = 0x0D; break;
			case '*': PORTB = 0x0E; break;
			case '0': PORTB = 0x00; break;
			case '#': PORTB = 0x0F; break;

			// Should never occur. Middle LED off.
			default: PORTB = 0x1B; break;
		}
	}
}
```

LCD Display:

LCD Display Pin Connections

LCD PIN	1	2	3	4	5	6	7-14	15-16
Connection	GND	5V	Potentio-meter (10KΩ) thru. to GND	AVR PORT A0	GND	AVR PORT A1	AVR PORT D0-D7	Vcc-GND

Here are the files required to run the LCD display. Make sure to change the values of DATA_BUS, CONTROL_BUS, RS, and E in io.c to reflect the new connections of the LCD screen.

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. We will define a structure called a *task* that represents a process in our operating system. The task structure should contain all of the information that represents that process, such as period, state, etc... The heart of each task is the function that it will be executing. 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 integer, but they have some specific syntax on how they must be called and defined. 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

Sample Task Scheduler:

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <ucr/bit.h>
#include <ucr/timer.h>
#include <stdio.h>

//--------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 ---------------------------

//--------Task scheduler data structure--------------------
// Struct for Tasks represent a running process in our
// simple real-time operating system.
/*Tasks should have members that include: state, period, a
measurement of elapsed time, and a function pointer.*/
typedef struct _task {

	//Task's current state, period, and the time elapsed
	// since the last tick
	signed char state;
	unsigned long int period;
	unsigned long int elapsedTime;

	//Task tick function
	int (*TickFct)(int);
} task;

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

//--------Shared Variables---------------------------------
unsigned char SM2_output = 0x00;
unsigned char SM3_output = 0x00;
unsigned char pause = 0;
//--------End Shared Variables-----------------------------
```

```c
//--------User defined FSMs--------------------------------
enum SM1_States { SM1_wait, SM1_press, SM1_release };

// Monitors button connected to PA0. When the button is
// pressed, shared variable "pause" is toggled.
int SMTick1(int state) {

	// Local Variables
	unsigned char press = ~PINA & 0x01;

	//State machine transitions
	switch (state) {

	// Wait for button press
	case SM1_wait:
		if (press == 0x01) { state = SM1_press; }
		break;

	// Button remains pressed
	case SM1_press:
		state = SM1_release;
		break;

	// Wait for button release
	case SM1_release:
		if (press == 0x00) { state = SM1_wait; }
		break;

	// default: Initial state
	default:
		state = SM1_wait;
		break;
	}

	//State machine actions
	switch(state) {
	case SM1_wait: break;
	case SM1_press:     // toggle pause
		pause = (pause == 0) ? 1 : 0;
		break;
	case SM1_release:   break;
	default:            break;
	}
	return state;
}
```

```c
enum SM2_States { SM2_wait, SM2_blink };

// If paused: Do NOT toggle LED connected to PB0
// If unpaused: toggle LED connected to PB0
int SMTick2(int state) {

	//State machine transitions
	switch (state) {
	case SM2_wait:

		// If unpaused, go to blink state
		if (pause == 0) { state = SM2_blink; }
		break;

	case SM2_blink:

		// If paused, go to wait state
		if (pause == 1) { state = SM2_wait; }
		break;

	default:
		state = SM2_wait;
		break;
	}

	//State machine actions
	switch(state) {
	case SM2_wait:
		break;

	//toggle LED
	case SM2_blink:
		SM2_output = (SM2_output == 0x00) ? 0x01 : 0x00;
		break;

	default:
		break;
	}

	return state;
}
```

```c
enum SM3_States { SM3_wait, SM3_blink };

// If paused: Do NOT toggle LED connected to PB1
// If unpaused: toggle LED connected to PB1
int SMTick3(int state) {

	//State machine transitions
	switch (state) {
	case SM3_wait:

		// If unpaused, go to blink state
		if (pause == 0) { state = SM3_blink; }
		break;

	case SM3_blink:

		// If paused, go to wait state
		if (pause == 1) { state = SM3_wait; }
		break;

	default:

		state = SM3_wait;
		break;
	}

	//State machine actions
	switch(state) {
	case SM3_wait:
		break;

	case SM3_blink:

		//toggle LED
		SM3_output = (SM3_output == 0x00) ? 0x02 : 0x00;
		break;

	default:
		break;
	}

	return state;
}
```

```c
enum SM4_States { SM4_display };

// Combine blinking LED outputs from SM2 and SM3, and
output on PORTB
int SMTick4(int state) {

	unsigned char output;

	//State machine transitions
	switch (state) {
	case SM4_display:
		break;

	default:
		state = SM4_display;
		break;
	}

	//State machine actions
	switch(state) {
	case SM4_display:

		// write shared outputs to local variables
		output = SM2_output | SM3_output;
		break;

	default:
		break;
	}

	// Write combined, shared output variables to PORTB
	PORTB = output;
	return state;
}

// --------END User defined FSMs---------------------------
```

```
// Implement scheduler code from PES.
int main()
{

    // Set Data Direction Registers
    // Buttons PORTA[0-7], set AVR PORTA
    // to pull down logic
    DDRA = 0x00; PORTA = 0xFF;
    DDRB = 0xFF; PORTB = 0x00;
    // . . . etc

    // Period for the tasks
    unsigned long int SMTick1_calc = 50;
    unsigned long int SMTick2_calc = 500;
    unsigned long int SMTick3_calc = 1000;
    unsigned long int SMTick4_calc = 10;

    //Calculating GCD
    unsigned long int tmpGCD = 1;
    tmpGCD = findGCD(SMTick1_calc, SMTick2_calc);
    tmpGCD = findGCD(tmpGCD, SMTick3_calc);
    tmpGCD = findGCD(tmpGCD, SMTick4_calc);

    //Greatest common divisor for all tasks
    // or smallest time unit for tasks.
    unsigned long int GCD = tmpGCD;

    //Recalculate GCD periods for scheduler
    unsigned long int SMTick1_period = SMTick1_calc/GCD;
    unsigned long int SMTick2_period = SMTick2_calc/GCD;
    unsigned long int SMTick3_period = SMTick3_calc/GCD;
    unsigned long int SMTick4_period = SMTick4_calc/GCD;

    //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
    task1.state = -1;
    task1.period = SMTick1_period;
    task1.elapsedTime = SMTick1_period;
    task1.TickFct = &SMTick1
```

```
    // Task 2
    task2.state = -1;
    task2.period = SMTick2_period;
    task2.elapsedTime = SMTick2_period;
    task2.TickFct = &SMTick2;

    // Task 3
    task3.state = -1;
    task3.period = SMTick3_period;
    task3.elapsedTime = SMTick3_period;
    task3.TickFct = &SMTick3;

    // Task 4
    task4.state = -1;
    task4.period = SMTick4_period;
    task4.elapsedTime = SMTick4_period;
    task4.TickFct = &SMTick4;

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

    // Scheduler for-loop iterator
    unsigned short i;
    while(1) {
        // Scheduler code
        for ( i = 0; i < numTasks; i++ ) {

            // Task is ready to tick
            if ( tasks[i]->elapsedTime ==
                tasks[i]->period ) {

                // Setting next state for task
                tasks[i]->state =
                    tasks[i]->TickFct(tasks[i]->state);

                // Reset elapsed time for next tick.
                tasks[i]->elapsedTime = 0;
            }
            tasks[i]->elapsedTime += 1;
        }
        while(!TimerFlag);
        TimerFlag = 0;
    }
    // Error: Program should not exit!
    return 0;
}
```

Pre-lab

Have your board wired up as above and ready to use (soldering of the LCD header must be completed before lab). Complete the GetKeyPad() function and be able to demo its fully working functionality (0 ~ 9, A ~ D, *, #). Be able to demo your LCD works.

Exercise 1

Modify the keypad code to be in an SM task. Then, modify the keypad SM to utilize the simple task scheduler format (refer to PES Chp 7). All code from here on out should use the task scheduler.

Exercise 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

Exercise 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

Exercise 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_BC9VuaIt8

Exercise 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:

- Use the cursor as the player controlled character.
- Choose a character like '#', '*', etc. to represent the obstacles.
- One button is used to pause/start the game.
- 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.
- A character position change should happen immediately after pressing the button.
- 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.
- Choose a reasonable movement speed for the obstacles (100ms or more).
- 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

Each student must submit an 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.