```
/*
 * adeja001_lab8_part1.c − April 8, 2013
 * Name: Ariana DeJaco  E−mail:adeja001@ucr.edu
 * CS Login: adeja001
 * Partner Name: Joshua DeForest−Williams jdefo002@ucr.edu
 * Lab Section: 022
 * Assignment: Lab#8 Exercise# Part1
 * Exercise Description: Rework the LCD and Keypad code to utilize the task scheduler format (Keypad code may continue to utilize a function call or may be modified into an SM task).  *       All code from here on out should use the task scheduler.
 */
#define F_CPU 8000000

#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/sfr_defs.h>
#include <util/delay.h>

////////////////////////////////////////////////////////////////////////////////
//Functionality − Sets bit on a PORTx
//Parameter: Takes in a uChar for a PORTx, the pin number and the binary value
//Returns: The new value of the PORTx
unsigned char SetBit(unsigned char pin, unsigned char number, unsigned char bin_value)
{
        return (bin_value ? pin | (0x01 << number) : pin & ~(0x01 << number));
}

////////////////////////////////////////////////////////////////////////////////
//Functionality − Gets bit from a PINx
//Parameter: Takes in a uChar for a PINx and the pin number
//Returns: The value of the PINx
unsigned char GetBit(unsigned char port, unsigned char number)
{
        return ( port & (0x01 << number) );
}

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 1ms
unsigned long _avr_timer_cntcurr = 0; // Current internal count of 1ms ticks

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

void TimerOn()
{
        // AVR timer/counter controller register TCCR0
        TCCR0   = 0x0B; // bit3bit6=10: CTC mode (clear timer on compare)
        // bit2bit1bit0=011: prescaler /64
        // 00001011: 0x0B
        // SO, 8 MHz clock or 8,000,000 /64 = 125,000 ticks/s
        // Thus, TCNT0 register will count at 125,000 ticks/s

        // AVR output compare register OCR0.
        OCR0    = 125;  // Timer interrupt will be generated when TCNT0==OCR0
        // We want a 1 ms tick. 0.001 s * 125,000 ticks/s = 125
        // So when TCNT0 register equals 125,
        // 1 ms has passed. Thus, we compare to 125.
        // AVR timer interrupt mask register

        TIMSK   = 0x02; // bit1: OCIE0 −− enables compare match interrupt

        //Initialize avr counter
        TCNT0 = 0;

```



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

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

void TimerOff()
{
        TCCR0   = 0x00; // bit2bit1bit0=000: timer off
}

void TimerISR() {
        TimerFlag = 1;
}

// In our approach, the C programmer does not touch this ISR, but rather TimerISR()
ISR(TIMER0_COMP_vect)
{
        // CPU automatically calls when TCNT0 == OCR0 (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;
        }
}

// 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
*/

/*****************************************************************************/
/* Set the LED to the value pressed in the KeyPad                            */
/*****************************************************************************/
void setValueToLEDs(unsigned char value)
{
        PORTA = (value << 2);
        // This is for active Low PORTA = (~(value << 2)) & 0xFC;
}

/*****************************************************************************/
/* Gets the value from the KeyPad                                            */
/*****************************************************************************/
unsigned char GetKeypadKey()
{
        PORTC = 0xEF; // Enable col 4 with 0, disable others with 1M−^Rs
        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 1M−^Rs
        asm("nop"); // add a delay to allow PORTC to stabilize before checking
        if (GetBit(PINC,0)==0) { return('2'); }
        if (GetBit(PINC,1)==0) { return('5'); }
        if (GetBit(PINC,2)==0) { return('8'); }
        if (GetBit(PINC,3)==0) { return('0'); }
```

```
138
139         // Check keys in col 3
140         PORTC = 0xBF; // Enable col 6 with 0, disable others with 1M−^Rs
141         asm("nop"); // add a delay to allow PORTC to stabilize before checking
142         if (GetBit(PINC,0)==0) { return('3'); }
143         if (GetBit(PINC,1)==0) { return('6'); }
144         if (GetBit(PINC,2)==0) { return('9'); }
145         if (GetBit(PINC,3)==0) { return('#'); }
146
147         // Check keys in col 4
148         PORTC = 0x7F;
149         asm("nop"); // add a delay to allow PORTC to stabilize before checking
150         if (GetBit(PINC,0)==0) { return('A'); }
151         if (GetBit(PINC,1)==0) { return('B'); }
152         if (GetBit(PINC,2)==0) { return('C'); }
153         if (GetBit(PINC,3)==0) { return('D'); }
154
155         return('\0'); // default value
156
157 }
158
159 /*********************************************************************/
160 /* Perform all of the initialization operations here.                */
161 /*********************************************************************/
162 void Init_KeyPad_Task()
163 {
164         DDRA = 0xFC; PORTA = 0x00; // PORTB set to output,Except for PA0 and PA2
    . PA0 is input switch.
165         DDRC = 0xF0; PORTC = 0x0F; // PC7..4 outputs init 0s, PC3..0 inputs init
     1s
166 }
167
168 /*********************************************************************/
169 /* Process_KeyPad_Task()                                             */
170 /* This method will advance the LCD task state machine one state per    */
171 /* timer tick.                                                       */
172 /*********************************************************************/
173 int Process_KeyPad_Task(int state)
174 {
175         unsigned char x;
176
177         x = GetKeypadKey();
178         switch (x)
179         {
180                 case '\0': setValueToLEDs(0x1F); break; // All 5 LEDs on
181                 case '1':  setValueToLEDs(0x01); break; // hex equivalent
182                 case '2':  setValueToLEDs(0x02); break;
183                 case '3':  setValueToLEDs(0x03); break;
184                 case '4':  setValueToLEDs(0x04); break;
185                 case '5':  setValueToLEDs(0x05); break;
186                 case '6':  setValueToLEDs(0x06); break;
187                 case '7':  setValueToLEDs(0x07); break;
188                 case '8':  setValueToLEDs(0x08); break;
189                 case '9':  setValueToLEDs(0x09); break;
190                 case 'A':  setValueToLEDs(0x0A); break;
191                 case 'B':  setValueToLEDs(0x0B); break;
192                 case 'C':  setValueToLEDs(0x0C); break;
193                 case 'D':  setValueToLEDs(0x0D); break;
194                 case '*':  setValueToLEDs(0x0E); break;
195                 case '0':  setValueToLEDs(0x00); break;
196                 case '#':  setValueToLEDs(0x0F); break;
197                 default:   setValueToLEDs(0x1B); break; // Should never occur. M
    iddle LED off.
198         }
199
200         // return state
201         return −1;
202 }
203
204 //********* LCD interface synchSM ********************************************
205
206 // Define LCD port assignments here so easier to change than if hardcoded below
207 unsigned char *LCD_Data = &PORTD;       // LCD 8−bit data bus
```



```
208 unsigned char *LCD_Ctrl = &PORTB;       // LCD needs 2−bits for control, use por
    t B
209 const unsigned char LCD_RS = 3;         // LCD Reset pin is PB3
210 const unsigned char LCD_E = 4;          // LCD Enable pin is PB4
211
212 unsigned char LCD_rdy_g = 0; // Set by LCD interface synchSM, ready to display n
    ew string
213 unsigned char LCD_go_g = 0; // Set by user synchSM wishing to display string in
    LCD_string_g
214 unsigned char LCD_string_g[17]; // Filled by user synchSM, 16 chars plus end−of−
    string char
215
216 void LCD_WriteCmdStart(unsigned char cmd) {
217         *LCD_Ctrl = SetBit(*LCD_Ctrl,LCD_RS, 0);
218         *LCD_Data = cmd;
219         *LCD_Ctrl = SetBit(*LCD_Ctrl,LCD_E, 1);
220 }
221 void LCD_WriteCmdEnd() {
222         *LCD_Ctrl = SetBit(*LCD_Ctrl,LCD_E, 0);
223 }
224 void LCD_WriteDataStart(unsigned char Data) {
225         *LCD_Ctrl = SetBit(*LCD_Ctrl,LCD_RS,1);
226         *LCD_Data = Data;
227         *LCD_Ctrl = SetBit(*LCD_Ctrl,LCD_E, 1);
228 }
229 void LCD_WriteDataEnd() {
230         *LCD_Ctrl = SetBit(*LCD_Ctrl,LCD_E, 0);
231 }
232 void LCD_Cursor(unsigned char column ) {
233         if ( column < 8 ) { // IEEE change this value to 16
234                 LCD_WriteCmdStart(0x80+column);
235         }
236         else {
237                 LCD_WriteCmdStart(0xB8+column); // IEEE change this value to 0xB
    F+column
238         }
239 }
240
241 enum LI_States { LI_Init1, LI_Init2, LI_Init3, LI_Init4, LI_Init5, LI_Init6,
242         LI_WaitDisplayString, LI_Clr, LI_PositionCursor, LI_DisplayChar, LI_Wait
    Go0 } LI_State;
243
244 void LI_Tick() {
245         static unsigned char i;
246         switch(LI_State) { // Transitions
247                 case −1:
248                         LI_State = LI_Init1;
249                         break;
250                 case LI_Init1:
251                         LI_State = LI_Init2;
252                         i=0;
253                         break;
254                 case LI_Init2:
255                         if (i<10) { // Wait 100 ms after power up
256                                 LI_State = LI_Init2;
257                         }
258                         else {
259                                 LI_State = LI_Init3;
260                         }
261                         break;
262                 case LI_Init3:
263                         LI_State = LI_Init4;
264                         LCD_WriteCmdEnd();
265                         break;
266                 case LI_Init4:
267                         LI_State = LI_Init5;
268                         LCD_WriteCmdEnd();
269                         break;
270                 case LI_Init5:
271                         LI_State = LI_Init6;
272                         LCD_WriteCmdEnd();
273                         break;
274                 case LI_Init6:
```

```
                        LI_State = LI_WaitDisplayString;
                        LCD_WriteCmdEnd();
                        break;
                ////////////////////////////////////////
                case LI_WaitDisplayString:
                        if (!LCD_go_g) {
                                LI_State = LI_WaitDisplayString;
                        }
                        else if (LCD_go_g) {
                         LCD_rdy_g = 0;
                                LI_State = LI_Clr;
                        }
                        break;
                case LI_Clr:
                        LI_State = LI_PositionCursor;
                        LCD_WriteCmdEnd();
                        i=0;
                        break;
                case LI_PositionCursor:
                        LI_State = LI_DisplayChar;
                        LCD_WriteCmdEnd();
                        break;
                case LI_DisplayChar:
                        if (i<16) {
                                LI_State = LI_PositionCursor;
                                LCD_WriteDataEnd();
                        i++;
                        }
                        else {
                                LI_State = LI_WaitGo0;
                                LCD_WriteDataEnd();
                        }
                        break;
                case LI_WaitGo0:
                        if (!LCD_go_g) {
                                LI_State = LI_WaitDisplayString;
                        }
                        else if (LCD_go_g) {
                                LI_State = LI_WaitGo0;
                        }
                        break;
                default:
                        LI_State = LI_Init1;
                } // Transitions

        switch(LI_State) { // State actions
                case LI_Init1:
                 LCD_rdy_g = 0;
                        break;
                case LI_Init2:
                        i++; // Waiting after power up
                        break;
                case LI_Init3:
                        LCD_WriteCmdStart(0x38);
                        break;
                case LI_Init4:
                        LCD_WriteCmdStart(0x06);
                        break;
                case LI_Init5:
                        LCD_WriteCmdStart(0x0F);
                        break;
                case LI_Init6:
                        LCD_WriteCmdStart(0x01); // Clear
                        break;
                ////////////////////////////////////////
                case LI_WaitDisplayString:
                        LCD_rdy_g = 1;
                        break;
                case LI_Clr:
                        LCD_WriteCmdStart(0x01);
                        break;
                case LI_PositionCursor:
                        LCD_Cursor(i);
```



```
                        break;
                case LI_DisplayChar:
                        LCD_WriteDataStart(LCD_string_g[i]);
                        break;
                case LI_WaitGo0:
                        break;
                default:
                        break;
        } // State actions
}
//--------END LCD interface synchSM-----------------------------------------
---


// SynchSM for testing the LCD interface -- waits for button press, fills LCD wi
th repeated random num

enum LT_States { LT_s0, LT_WaitLcdRdy, LT_WaitButton, LT_FillAndDispString,
LT_HoldGo1, LT_WaitBtnRelease } LT_State;

void LT_Tick() {
        static unsigned char i, x, c;
        switch(LT_State) { // Transitions
                case -1:
                        LT_State = LT_s0;
                        break;
                case LT_s0:
                        LT_State = LT_WaitLcdRdy;
                        break;
                case LT_WaitLcdRdy:
                        if (!LCD_rdy_g) {
                                LT_State = LT_WaitLcdRdy;
                        }
                        else if (LCD_rdy_g) {
                                LT_State = LT_WaitButton;
                        }
                        break;
                case LT_WaitButton:
                        if (GetBit(PINA,0)==1) {
                                LT_State = LT_WaitButton;
                        }
                        else if (GetBit(PINA,0)==0) { // Button active low
                                LT_State = LT_FillAndDispString;
                        }
                        break;
                case LT_FillAndDispString:
                        LT_State = LT_HoldGo1;
                        break;
                case LT_HoldGo1:
                        LCD_go_g=0;
                        LT_State = LT_WaitBtnRelease;
                        break;
                case LT_WaitBtnRelease:
                        if (GetBit(PINA,0)==0) { // Wait for button release
                                LT_State = LT_WaitBtnRelease;
                        }
                        else if (GetBit(PINA,0)==1) {
                                LT_State = LT_WaitLcdRdy;
                        }
                        break;
                default:
                        LT_State = LT_s0;
                } // Transitions

        switch(LT_State) { // State actions
                case LT_s0:
                        LCD_go_g=0;
                        strcpy(LCD_string_g, "1234567890123456"); // Init, but nev
er seen, shows use of strcpy though
                        break;
                case LT_WaitLcdRdy:
                        break;
                case LT_WaitButton:
```

```
418                         break;
419                 case LT_FillAndDispString:
420                         x = rand() % 10; // x is rand num 0-9
421                         c = (char)(((int)'0')+x); // c is ascii of num x
422                         for (i=0; i<16; i++) { // Fill string with c
423                               LCD_string_g[i] = c;
424                         }
425                         LCD_string_g[i] = '\0'; // End-of-string char
426                         LCD_go_g = 1; // Display string
427                         break;
428                 case LT_HoldGo1:
429                         break;
430                 case LT_WaitBtnRelease:
431                         break;
432                 default:
433                         break;
434         } // State actions
435 }
436
437 /*****************************************************************************/
438 /* Perform all of the initialization operations here.                        */
439 /*****************************************************************************/
440 void Init_LCD_Task()
441 {
442         DDRB = 0xFF; // Set port B to output
443         DDRD = 0xFF; // Set port D to output
444         DDRA = 0xFC; // PA0 is an input
445         LI_State = -1;
446         LT_State = -1;
447 }
448
449 /*****************************************************************************/
450 /* Process_LCD_Task()                                                        */
451 /* This method will advance the LCD task state machine one state per         */
452 /* timer tick.                                                               */
453 /*****************************************************************************/
454 int Process_LCD_Task(int state)
455 {
456         LI_Tick();
457         LT_Tick();
458
459         // Don't really needed to pass state information to the scheduler
460         return 1;
461 }
462
463 ////////////////////////////////////////////////////////////////////////////////
464 //Functionality - finds the greatest common divisor of two values
465 //Parameter: Two long int's to find their GCD
466 //Returns: GCD else 0
467 unsigned long int findGCD(unsigned long int a, unsigned long int b)
468 {
469         unsigned long int c;
470         while(1){
471                 c = a % b;
472         if( c == 0 ) { return b; }
473         a = b;
474         b = c;
475 }
476 return 0;
477 }
478
479 ////////////////////////////////////////////////////////////////////////////////
480 //Struct for Tasks represent a running process in our simple real-time operating
     system
481 typedef struct _task
482 {
483         // Tasks should have members that include: state, period,
484         //a measurement of elapsed time, and a function pointer.
485         signed   char state;           //Task's current state
486         unsigned long period;          //Task period
487         unsigned long elapsedTime;     //Time elapsed since last task tick
488         int (*TickFct)(int);           //Task tick function
489 } task;
```



```
490
491 // Implement scheduler code from PES.
492 void RunTaskScheduler()
493 {
494         // Initialize the tasks.
495         Init_KeyPad_Task();
496         Init_LCD_Task();
497
498         unsigned long int LCD_Tick    = 10;
499         unsigned long int KeyPad_Tick = 50;
500
501         // Calculate the GCD of the LCD and KeyPad tasks
502         unsigned long int tmpGCD = findGCD(LCD_Tick, KeyPad_Tick);
503
504         //Greatest common divisor for all tasks or smallest time unit for tasks.
505         unsigned long int GCD = tmpGCD;
506
507         //Recalculate GCD periods for scheduler
508         unsigned long int LCD_period    = LCD_Tick/GCD;
509         unsigned long int KeyPad_period = KeyPad_Tick/GCD;
510
511         // Declare an array of tasks
512         static task task1, task2;
513         task *tasks[] = { &task1, &task2};
514         const unsigned short numTasks = sizeof(tasks)/sizeof(task*);
515
516         // Task 1
517         task1.state = -1;//Task initial state.
518         task1.period = LCD_period;//Task Period.
519         task1.elapsedTime = LCD_period;//Task current elapsed time.
520         task1.TickFct = &Process_LCD_Task;//Function pointer for the tick.
521
522         // Task 2
523         task2.state = -1;//Task initial state.
524         task2.period = KeyPad_period;//Task Period.
525         task2.elapsedTime = KeyPad_period;//Task current elapsed time.
526         task2.TickFct = &Process_KeyPad_Task;//Function pointer for the tick.
527
528         // Set the timer and turn it on
529         TimerSet(GCD);
530         TimerOn();
531
532         // Run all tasks forever....
533         unsigned short i; // Scheduler for-loop iterator
534         while(1)
535         {
536                 // Scheduler code
537                 for ( i = 0; i < 2; i++ )
538                 {
539                         // Task is ready to tick
540                         if ( tasks[i]->elapsedTime == tasks[i]->period )
541                         {
542                                 // Setting next state for task
543                                 tasks[i]->state = tasks[i]->TickFct(tasks[i]->st
    ate);
544                                 // Reset the elapsed time for next tick.
545                                 tasks[i]->elapsedTime = 0;
546                         }
547                         tasks[i]->elapsedTime += 1;
548                 }
549
550                 // Process_LCD_Task(1);
551                 while(!TimerFlag);
552                 TimerFlag = 0;
553         }
554 }
555
556
557 int main(void)
558 {
559         // The Task scheduler does all of the real work
560         RunTaskScheduler();
561 }
```

```
/*
 * adeja001_lab8_part2.c − April 8, 2013
 * Name: Ariana DeJaco  E−mail:adeja001@ucr.edu
 * CS Login: adeja001
 * Partner Name: Joshua DeForest−Williams jdefo002@ucr.edu
 * Lab Section: 022
 * Assignment: Lab#8 Exercise# Part2
 * Exercise 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.
 */
#define F_CPU 8000000

#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/sfr_defs.h>
#include <util/delay.h>

////////////////////////////////////////////////////////////////////////////////
//Functionality − Sets bit on a PORTx
//Parameter: Takes in a uChar for a PORTx, the pin number and the binary value
//Returns: The new value of the PORTx
unsigned char SetBit(unsigned char pin, unsigned char number, unsigned char bin_value)
{
        return (bin_value ? pin | (0x01 << number) : pin & ~(0x01 << number));
}

////////////////////////////////////////////////////////////////////////////////
//Functionality − Gets bit from a PINx
//Parameter: Takes in a uChar for a PINx and the pin number
//Returns: The value of the PINx
unsigned char GetBit(unsigned char port, unsigned char number)
{
        return ( port & (0x01 << number) );
}

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 1ms
unsigned long _avr_timer_cntcurr = 0; // Current internal count of 1ms ticks

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

void TimerOn()
{
        // AVR timer/counter controller register TCCR0
        TCCR0   = 0x0B; // bit3bit6=10: CTC mode (clear timer on compare)
        // bit2bit1bit0=011: prescaler /64
        // 00001011: 0x0B
        // SO, 8 MHz clock or 8,000,000 /64 = 125,000 ticks/s
        // Thus, TCNT0 register will count at 125,000 ticks/s

        // AVR output compare register OCR0.
        OCR0    = 125;  // Timer interrupt will be generated when TCNT0==OCR0
        // We want a 1 ms tick. 0.001 s * 125,000 ticks/s = 125
        // So when TCNT0 register equals 125,
        // 1 ms has passed. Thus, we compare to 125.
        // AVR timer interrupt mask register

        TIMSK   = 0x02; // bit1: OCIE0 −− enables compare match interrupt

        //Initialize avr counter
        TCNT0 = 0;

```



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

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

void TimerOff()
{
        TCCR0   = 0x00; // bit2bit1bit0=000: timer off
}

void TimerISR() {
        TimerFlag = 1;
}

// In our approach, the C programmer does not touch this ISR, but rather TimerISR()
ISR(TIMER0_COMP_vect)
{
        // CPU automatically calls when TCNT0 == OCR0 (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;
        }
}

// 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
*/

/*****************************************************************************/
/* Set the LED to the value pressed in the KeyPad                            */
/*****************************************************************************/
void setValueToLEDs(unsigned char value)
{
        PORTA = (value << 2);
        // This is for active Low PORTA = (~(value << 2)) & 0xFC;
}

/*****************************************************************************/
/* Gets the value from the KeyPad                                            */
/*****************************************************************************/
unsigned char GetKeypadKey()
{
        PORTC = 0xEF; // Enable col 4 with 0, disable others with 1M−^Rs
        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 1M−^Rs
        asm("nop"); // add a delay to allow PORTC to stabilize before checking
        if (GetBit(PINC,0)==0) { return('2'); }
        if (GetBit(PINC,1)==0) { return('5'); }
        if (GetBit(PINC,2)==0) { return('8'); }
        if (GetBit(PINC,3)==0) { return('0'); }
```

```
138
139         // Check keys in col 3
140         PORTC = 0xBF; // Enable col 6 with 0, disable others with 1M−^Rs
141         asm("nop"); // add a delay to allow PORTC to stabilize before checking
142         if (GetBit(PINC,0)==0) { return('3'); }
143         if (GetBit(PINC,1)==0) { return('6'); }
144         if (GetBit(PINC,2)==0) { return('9'); }
145         if (GetBit(PINC,3)==0) { return('#'); }
146
147         // Check keys in col 4
148         PORTC = 0x7F;
149         asm("nop"); // add a delay to allow PORTC to stabilize before checking
150         if (GetBit(PINC,0)==0) { return('A'); }
151         if (GetBit(PINC,1)==0) { return('B'); }
152         if (GetBit(PINC,2)==0) { return('C'); }
153         if (GetBit(PINC,3)==0) { return('D'); }
154
155         return('\0'); // default value
156
157 }
158
159 /*******************************************************************/
160 /* Perform all of the initialization operations here.              */
161 /*******************************************************************/
162 void Init_KeyPad_Task()
163 {
164         DDRA = 0xFC; PORTA = 0x00; // PORTB set to output,Except for PA0 and PA2
    . PA0 is input switch.
165         DDRC = 0xF0; PORTC = 0x0F; // PC7..4 outputs init 0s, PC3..0 inputs init
     1s
166 }
167
168 /*******************************************************************/
169 /* Process_KeyPad_Task()                                           */
170 /* This method will advance the LCD task state machine one state per    */
171 /* timer tick.                                                     */
172 /*******************************************************************/
173 int Process_KeyPad_Task(int state)
174 {
175         unsigned char x;
176
177         x = GetKeypadKey();
178         switch (x)
179         {
180                 case '\0': setValueToLEDs(0x1F); break; // All 5 LEDs on
181                 case '1':  setValueToLEDs(0x01); break; // hex equivalent
182                 case '2':  setValueToLEDs(0x02); break;
183                 case '3':  setValueToLEDs(0x03); break;
184                 case '4':  setValueToLEDs(0x04); break;
185                 case '5':  setValueToLEDs(0x05); break;
186                 case '6':  setValueToLEDs(0x06); break;
187                 case '7':  setValueToLEDs(0x07); break;
188                 case '8':  setValueToLEDs(0x08); break;
189                 case '9':  setValueToLEDs(0x09); break;
190                 case 'A':  setValueToLEDs(0x0A); break;
191                 case 'B':  setValueToLEDs(0x0B); break;
192                 case 'C':  setValueToLEDs(0x0C); break;
193                 case 'D':  setValueToLEDs(0x0D); break;
194                 case '*':  setValueToLEDs(0x0E); break;
195                 case '0':  setValueToLEDs(0x00); break;
196                 case '#':  setValueToLEDs(0x0F); break;
197                 default:   setValueToLEDs(0x1B); break; // Should never occur. M
    iddle LED off.
198         }
199
200         // return state
201         return −1;
202 }
203
204 //********* LCD interface synchSM ********************************************
205
206 // Define LCD port assignments here so easier to change than if hardcoded below
207 unsigned char *LCD_Data = &PORTD;       // LCD 8−bit data bus
```



```
208 unsigned char *LCD_Ctrl = &PORTB;       // LCD needs 2−bits for control, use por
    t B
209 const unsigned char LCD_RS = 3;         // LCD Reset pin is PB3
210 const unsigned char LCD_E = 4;          // LCD Enable pin is PB4
211
212 unsigned char LCD_rdy_g = 0; // Set by LCD interface synchSM, ready to display n
    ew string
213 unsigned char LCD_go_g = 0; // Set by user synchSM wishing to display string in
    LCD_string_g
214 unsigned char LCD_string_g[17]; // Filled by user synchSM, 16 chars plus end−of−
    string char
215
216 void LCD_WriteCmdStart(unsigned char cmd) {
217         *LCD_Ctrl = SetBit(*LCD_Ctrl,LCD_RS, 0);
218         *LCD_Data = cmd;
219         *LCD_Ctrl = SetBit(*LCD_Ctrl,LCD_E, 1);
220 }
221 void LCD_WriteCmdEnd() {
222         *LCD_Ctrl = SetBit(*LCD_Ctrl,LCD_E, 0);
223 }
224 void LCD_WriteDataStart(unsigned char Data) {
225         *LCD_Ctrl = SetBit(*LCD_Ctrl,LCD_RS,1);
226         *LCD_Data = Data;
227         *LCD_Ctrl = SetBit(*LCD_Ctrl,LCD_E, 1);
228 }
229 void LCD_WriteDataEnd() {
230         *LCD_Ctrl = SetBit(*LCD_Ctrl,LCD_E, 0);
231 }
232 void LCD_Cursor(unsigned char column ) {
233         if ( column < 8 ) { // IEEE change this value to 16
234                 LCD_WriteCmdStart(0x80+column);
235         }
236         else {
237                 LCD_WriteCmdStart(0xB8+column); // IEEE change this value to 0xB
    F+column
238         }
239 }
240
241 enum LI_States { Initial, LI_Init1, LI_Init2, LI_Init3, LI_Init4, LI_Init5, LI_I
    nit6,
242         LI_WaitDisplayString, LI_Clr, LI_PositionCursor, LI_DisplayChar, LI_Wait
    Go0 } LI_State;
243
244 void LI_Tick() {
245         static unsigned char i;
246         switch(LI_State) { // Transitions
247                 case Initial:
248                         LI_State = LI_Init1;
249                         break;
250                 case LI_Init1:
251                         LI_State = LI_Init2;
252                         i=0;
253                         break;
254                 case LI_Init2:
255                         if (i<10) { // Wait 100 ms after power up
256                                 LI_State = LI_Init2;
257                         }
258                         else {
259                                 LI_State = LI_Init3;
260                         }
261                         break;
262                 case LI_Init3:
263                         LI_State = LI_Init4;
264                         LCD_WriteCmdEnd();
265                         break;
266                 case LI_Init4:
267                         LI_State = LI_Init5;
268                         LCD_WriteCmdEnd();
269                         break;
270                 case LI_Init5:
271                         LI_State = LI_Init6;
272                         LCD_WriteCmdEnd();
273                         break;
```

```
274                     case LI_Init6:
275                             LI_State = LI_WaitDisplayString;
276                             LCD_WriteCmdEnd();
277                             break;
278                     ////////////////////////////////////////////
279                     case LI_WaitDisplayString:
280                             if (!LCD_go_g) {
281                                     LI_State = LI_WaitDisplayString;
282                             }
283                             else if (LCD_go_g) {
284                              LCD_rdy_g = 0;
285                                     LI_State = LI_Clr;
286                             }
287                             break;
288                     case LI_Clr:
289                             LI_State = LI_PositionCursor;
290                             LCD_WriteCmdEnd();
291                             i=0;
292                             break;
293                     case LI_PositionCursor:
294                             LI_State = LI_DisplayChar;
295                             LCD_WriteCmdEnd();
296                             break;
297                     case LI_DisplayChar:
298                             if (i<16) {
299                                     LI_State = LI_PositionCursor;
300                                     LCD_WriteDataEnd();
301                             i++;
302                             }
303                             else {
304                                     LI_State = LI_WaitGo0;
305                                     LCD_WriteDataEnd();
306                             }
307                             break;
308                     case LI_WaitGo0:
309                             if (!LCD_go_g) {
310                                     LI_State = LI_WaitDisplayString;
311                             }
312                             else if (LCD_go_g) {
313                                     LI_State = LI_WaitGo0;
314                             }
315                             break;
316                     default:
317                             LI_State = LI_Init1;
318                     } // Transitions
319
320             switch(LI_State) { // State actions
321                     case LI_Init1:
322                      LCD_rdy_g = 0;
323                             break;
324                     case LI_Init2:
325                             i++; // Waiting after power up
326                             break;
327                     case LI_Init3:
328                             LCD_WriteCmdStart(0x38);
329                             break;
330                     case LI_Init4:
331                             LCD_WriteCmdStart(0x06);
332                             break;
333                     case LI_Init5:
334                             LCD_WriteCmdStart(0x0F);
335                             break;
336                     case LI_Init6:
337                             LCD_WriteCmdStart(0x01); // Clear
338                             break;
339                     ////////////////////////////////////////////
340                     case LI_WaitDisplayString:
341                             LCD_rdy_g = 1;
342                             break;
343                     case LI_Clr:
344                             LCD_WriteCmdStart(0x01);
345                             break;
346                     case LI_PositionCursor:
```



```
347                             LCD_Cursor(i);
348                             break;
349                     case LI_DisplayChar:
350                             LCD_WriteDataStart(LCD_string_g[i]);
351                             break;
352                     case LI_WaitGo0:
353                             break;
354                     default:
355                             break;
356             } // State actions
357     }
358     //--------END LCD interface synchSM-----------------------------------------
        ---
359
360
361     // SynchSM for testing the LCD interface -- waits for button press, fills LCD wi
        th repeated random num
362
363     enum LT_States { Initial1, LT_s0, LT_WaitLcdRdy, LT_WaitButton, LT_FillAndDispSt
        ringPartOne,
364     LT_FillAndDispStringPartTwo, LT_HoldGo1, LT_WaitBtnRelease } LT_State;
365
366     void LT_Tick() {
367             //static unsigned char i, x, c;
368             const char* sentencePartOne = "CS120B is Legend...";
369             const char* sentencePartTwo = "wait for it DARY!";
370             static unsigned char previousState = 0;
371             switch(LT_State) { // Transitions
372                     case Initial1:
373                             LT_State = LT_s0;
374                             break;
375                     case LT_s0:
376                             LT_State = LT_WaitLcdRdy;
377                             break;
378                     case LT_WaitLcdRdy:
379                             if (!LCD_rdy_g) {
380                                     LT_State = LT_WaitLcdRdy;
381                             }
382                             else if (LCD_rdy_g) {
383                                     LT_State = LT_WaitButton;
384                             }
385                             break;
386                     case LT_WaitButton:
387                             if (GetBit(PINA,0)==1) {
388                                     LT_State = LT_WaitButton;
389                             }
390                             else if (GetBit(PINA,0)==0 && ((previousState == 0) || (
        previousState == 2)) ) { // Button active low
391                                     LT_State = LT_FillAndDispStringPartOne;
392                             }
393                             else if (GetBit(PINA,0)==0 && (previousState == 1) ) { /
        / Button active low
394                             LT_State = LT_FillAndDispStringPartTwo;
395                             }
396                             break;
397                     case LT_FillAndDispStringPartOne:
398                             LT_State = LT_HoldGo1;
399                             previousState = 1;
400                             break;
401                     case LT_FillAndDispStringPartTwo:
402                             LT_State = LT_HoldGo1;
403                             previousState = 2;
404                             break;
405                     case LT_HoldGo1:
406                             LCD_go_g=0;
407                             LT_State = LT_WaitBtnRelease;
408                             break;
409                     case LT_WaitBtnRelease:
410                             if (GetBit(PINA,0)==0) { // Wait for button release
411                                     LT_State = LT_WaitBtnRelease;
412                             }
413                             else if (GetBit(PINA,0)==1) {
414                                     LT_State = LT_WaitLcdRdy;
```

```
415                         }
416                         break;
417                 default:
418                         LT_State = LT_s0;
419                 } // Transitions
420
421         switch(LT_State) { // State actions
422                 case LT_s0:
423                         LCD_go_g=0;
424                         strcpy(LCD_string_g, "1234567890123456"); // Init, but nev
    er seen, shows use of strcpy though
425                         break;
426                 case LT_WaitLcdRdy:
427                         break;
428                 case LT_WaitButton:
429                         break;
430                 case LT_FillAndDispStringPartOne:
431                         strcpy(LCD_string_g, sentencePartOne);
432                         LCD_string_g[strlen(sentencePartOne)] = '\0'; // End-of-
    string char
433                         LCD_go_g = 1; // Display string
434                         break;
435                 case LT_FillAndDispStringPartTwo:
436                         strcpy(LCD_string_g, sentencePartTwo);
437                         LCD_string_g[strlen(sentencePartTwo)] = '\0'; // End-of-
    string char
438                         LCD_go_g = 1; // Display string
439                         break;
440                 case LT_HoldGo1:
441                         break;
442                 case LT_WaitBtnRelease:
443                         break;
444                 default:
445                         break;
446         } // State actions
447 }
448
449 /*****************************************************************************/
450 /* Perform all of the initialization operations here.                        */
451 /*****************************************************************************/
452 void Init_LCD_Task()
453 {
454         DDRB = 0xFF; // Set port B to output
455         DDRD = 0xFF; // Set port D to output
456         DDRA = 0xFC; // PA0 is an input
457         LI_State = -1;
458         LT_State = -1;
459 }
460
461 /*****************************************************************************/
462 /* Process_LCD_Task()                                                        */
463 /* This method will advance the LCD task state machine one state per         */
464 /* timer tick.                                                               */
465 /*****************************************************************************/
466 int Process_LCD_Task(int state)
467 {
468         LI_Tick();
469         LT_Tick();
470
471         // Don't really needed to pass state information to the scheduler
472         return 1;
473 }
474
475 ////////////////////////////////////////////////////////////////////////////////
476 //Functionality - finds the greatest common divisor of two values
477 //Parameter: Two long int's to find their GCD
478 //Returns: GCD else 0
479 unsigned long int findGCD(unsigned long int a, unsigned long int b)
480 {
481         unsigned long int c;
482         while(1){
483                 c = a % b;
484         if( c == 0 ) { return b; }
```



```
485         a = b;
486         b = c;
487         }
488         return 0;
489 }
490
491 ////////////////////////////////////////////////////////////////////////////////
492 //Struct for Tasks represent a running process in our simple real-time operating
     system
493 typedef struct _task
494 {
495         // Tasks should have members that include: state, period,
496         //a measurement of elapsed time, and a function pointer.
497         signed   char state;            //Task's current state
498         unsigned long period;           //Task period
499         unsigned long elapsedTime;      //Time elapsed since last task tick
500         int (*TickFct)(int);            //Task tick function
501 } task;
502
503 // Implement scheduler code from PES.
504 void RunTaskScheduler()
505 {
506         // Initialize the tasks.
507 //      Init_KeyPad_Task();
508         Init_LCD_Task();
509
510         unsigned long int LCD_Tick    = 20;
511 //      unsigned long int KeyPad_Tick = 100;
512
513         // Calculate the GCD of the LCD and KeyPad tasks
514 //      unsigned long int tmpGCD = findGCD(LCD_Tick, KeyPad_Tick);
515
516         //Greatest common divisor for all tasks or smallest time unit for tasks.
517         unsigned long int GCD = LCD_Tick; //tmpGCD;
518
519         //Recalculate GCD periods for scheduler
520         unsigned long int LCD_period    = LCD_Tick/GCD;
521 //      unsigned long int KeyPad_period = KeyPad_Tick/GCD;
522
523         // Declare an array of tasks
524         static task task1; //, task2;
525 //      task *tasks[] = { &task1, &task2};
526 //      const unsigned short numTasks = 1; //sizeof(tasks)/sizeof(task*);
527
528         // Task 1
529         task1.state = -1;//Task initial state.
530         task1.period = LCD_period;//Task Period.
531         task1.elapsedTime = LCD_period;//Task current elapsed time.
532         task1.TickFct = &Process_LCD_Task;//Function pointer for the tick.
533
534         // Task 2
535 /*      task2.state = -1;//Task initial state.
536         task2.period = KeyPad_period;//Task Period.
537         task2.elapsedTime = KeyPad_period;//Task current elapsed time.
538         task2.TickFct = &Process_KeyPad_Task;//Function pointer for the tick.
539 */
540         // Set the timer and turn it on
541         TimerSet(GCD);
542         TimerOn();
543
544         // Run all tasks forever....
545 //      unsigned short i; // Scheduler for-loop iterator
546         while(1)
547         {
548                 // Scheduler code
549 //              for ( i = 0; i < 2; i++ )
550 //              {
551                         // Task is ready to tick
552                         if ( task1.elapsedTime == task1.period )
553                         {
554                                 // Setting next state for task
555                                 task1.state = task1.TickFct(task1.state);
556                                 // Reset the elapsed time for next tick.
```

```
557                                     task1.elapsedTime = 0;
558                             }
559                             task1.elapsedTime += 1;
560             //      }
561
562                     // Process_LCD_Task(1);
563                     while(!TimerFlag);
564                     TimerFlag = 0;
565             }
566     }
567
568
569     int main(void)
570     {
571             // The Task scheduler does all of the real work
572             RunTaskScheduler();
573     }
```

```
/*
 * adeja001_lab8_part3.c − April 8, 2013
 * Name: Ariana DeJaco  E−mail:adeja001@ucr.edu
 * CS Login: adeja001
 * Partner Name: Joshua DeForest−Williams jdefo002@ucr.edu
 * Lab Section: 022
 * Assignment: Lab#8 Exercise#3
 * Exercise Description: 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).
 */
#define F_CPU 8000000

#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/sfr_defs.h>
#include <util/delay.h>

////////////////////////////////////////////////////////////////////////////////
//Functionality − Sets bit on a PORTx
//Parameter: Takes in a uChar for a PORTx, the pin number and the binary value
//Returns: The new value of the PORTx
unsigned char SetBit(unsigned char pin, unsigned char number, unsigned char bin_value)
{
        return (bin_value ? pin | (0x01 << number) : pin & ~(0x01 << number));
}

////////////////////////////////////////////////////////////////////////////////
//Functionality − Gets bit from a PINx
//Parameter: Takes in a uChar for a PINx and the pin number
//Returns: The value of the PINx
unsigned char GetBit(unsigned char port, unsigned char number)
{
        return ( port & (0x01 << number) );
}

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 1ms
unsigned long _avr_timer_cntcurr = 0; // Current internal count of 1ms ticks

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

void TimerOn()
{
        // AVR timer/counter controller register TCCR0
        TCCR0   = 0x0B; // bit3bit6=10: CTC mode (clear timer on compare)
        // bit2bit1bit0=011: prescaler /64
        // 00001011: 0x0B
        // SO, 8 MHz clock or 8,000,000 /64 = 125,000 ticks/s
        // Thus, TCNT0 register will count at 125,000 ticks/s

        // AVR output compare register OCR0.
        OCR0    = 125;  // Timer interrupt will be generated when TCNT0==OCR0
        // We want a 1 ms tick. 0.001 s * 125,000 ticks/s = 125
        // So when TCNT0 register equals 125,
        // 1 ms has passed. Thus, we compare to 125.
        // AVR timer interrupt mask register

        TIMSK   = 0x02; // bit1: OCIE0 −− enables compare match interrupt

        //Initialize avr counter
        TCNT0 = 0;
```



```

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

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

void TimerOff()
{
        TCCR0   = 0x00; // bit2bit1bit0=000: timer off
}

void TimerISR() {
        TimerFlag = 1;
}

// In our approach, the C programmer does not touch this ISR, but rather TimerISR()
ISR(TIMER0_COMP_vect)
{
        // CPU automatically calls when TCNT0 == OCR0 (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;
        }
}

// 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
*/

/*****************************************************************************/
/* Set the LED to the value pressed in the KeyPad                            */
/*****************************************************************************/
void setValueToLEDs(unsigned char value)
{
        PORTA = (value << 2);
        // This is for active Low PORTA = (~(value << 2)) & 0xFC;
}

/*****************************************************************************/
/* Gets the value from the KeyPad                                            */
/*****************************************************************************/
unsigned char GetKeypadKey()
{
        PORTC = 0xEF; // Enable col 4 with 0, disable others with 1M−^Rs
        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 1M−^Rs
        asm("nop"); // add a delay to allow PORTC to stabilize before checking
        if (GetBit(PINC,0)==0) { return('2'); }
        if (GetBit(PINC,1)==0) { return('5'); }
        if (GetBit(PINC,2)==0) { return('8'); }
```

```
138         if (GetBit(PINC,3)==0) { return('0'); }
139
140         // Check keys in col 3
141         PORTC = 0xBF; // Enable col 6 with 0, disable others with 1M−^Rs
142         asm("nop"); // add a delay to allow PORTC to stabilize before checking
143         if (GetBit(PINC,0)==0) { return('3'); }
144         if (GetBit(PINC,1)==0) { return('6'); }
145         if (GetBit(PINC,2)==0) { return('9'); }
146         if (GetBit(PINC,3)==0) { return('#'); }
147
148         // Check keys in col 4
149         PORTC = 0x7F;
150         asm("nop"); // add a delay to allow PORTC to stabilize before checking
151         if (GetBit(PINC,0)==0) { return('A'); }
152         if (GetBit(PINC,1)==0) { return('B'); }
153         if (GetBit(PINC,2)==0) { return('C'); }
154         if (GetBit(PINC,3)==0) { return('D'); }
155
156         return('\0'); // default value
157
158 }
159
160 /*****************************************************************************/
161 /* Perform all of the initialization operations here.                        */
162 /*****************************************************************************/
163 void Init_KeyPad_Task()
164 {
165         DDRA = 0xFC; PORTA = 0x00; // PORTB set to output,Except for PA0 and PA2
    . PA0 is input switch.
166         DDRC = 0xF0; PORTC = 0x0F; // PC7..4 outputs init 0s, PC3..0 inputs init
     1s
167 }
168
169 /*****************************************************************************/
170 /* Process_KeyPad_Task()                                                     */
171 /* This method will advance the LCD task state machine one state per         */
172 /* timer tick.                                                               */
173 /*****************************************************************************/
174 unsigned char keypad_number = 0;
175 unsigned char prev_key_num = 0;
176 int Process_KeyPad_Task(int state)
177 {
178         unsigned char x;
179    prev_key_num = keypad_number;
180         x = GetKeypadKey();
181    keypad_number = x;
182         switch (x)
183         {
184                 case '\0': setValueToLEDs(0x1F); break; // All 5 LEDs on
185                 case '1':  setValueToLEDs(0x01); break; // hex equivalent
186                 case '2':  setValueToLEDs(0x02); break;
187                 case '3':  setValueToLEDs(0x03); break;
188                 case '4':  setValueToLEDs(0x04); break;
189                 case '5':  setValueToLEDs(0x05); break;
190                 case '6':  setValueToLEDs(0x06); break;
191                 case '7':  setValueToLEDs(0x07); break;
192                 case '8':  setValueToLEDs(0x08); break;
193                 case '9':  setValueToLEDs(0x09); break;
194                 case 'A':  setValueToLEDs(0x0A); break;
195                 case 'B':  setValueToLEDs(0x0B); break;
196                 case 'C':  setValueToLEDs(0x0C); break;
197                 case 'D':  setValueToLEDs(0x0D); break;
198                 case '*':  setValueToLEDs(0x0E); break;
199                 case '0':  setValueToLEDs(0x00); break;
200                 case '#':  setValueToLEDs(0x0F); break;
201                 default:   setValueToLEDs(0x1B); break; // Should never occur. M
    iddle LED off.
202         }
203
204         // return state
205         return −1;
206 }
207
```



```
208 //********* LCD interface synchSM ********************************************
209
210 // Define LCD port assignments here so easier to change than if hardcoded below
211 unsigned char *LCD_Data = &PORTD;       // LCD 8−bit data bus
212 unsigned char *LCD_Ctrl = &PORTB;       // LCD needs 2−bits for control, use por
    t B
213 const unsigned char LCD_RS = 3;         // LCD Reset pin is PB3
214 const unsigned char LCD_E = 4;          // LCD Enable pin is PB4
215
216 unsigned char LCD_rdy_g = 0; // Set by LCD interface synchSM, ready to display n
    ew string
217 unsigned char LCD_go_g = 0; // Set by user synchSM wishing to display string in
    LCD_string_g
218 unsigned char LCD_string_g[17]; // Filled by user synchSM, 16 chars plus end−of−
    string char
219
220 void LCD_WriteCmdStart(unsigned char cmd) {
221         *LCD_Ctrl = SetBit(*LCD_Ctrl,LCD_RS, 0);
222         *LCD_Data = cmd;
223         *LCD_Ctrl = SetBit(*LCD_Ctrl,LCD_E, 1);
224 }
225 void LCD_WriteCmdEnd() {
226         *LCD_Ctrl = SetBit(*LCD_Ctrl,LCD_E, 0);
227 }
228 void LCD_WriteDataStart(unsigned char Data) {
229         *LCD_Ctrl = SetBit(*LCD_Ctrl,LCD_RS,1);
230         *LCD_Data = Data;
231         *LCD_Ctrl = SetBit(*LCD_Ctrl,LCD_E, 1);
232 }
233 void LCD_WriteDataEnd() {
234         *LCD_Ctrl = SetBit(*LCD_Ctrl,LCD_E, 0);
235 }
236 void LCD_Cursor(unsigned char column ) {
237         if ( column < 8 ) { // IEEE change this value to 16
238                 LCD_WriteCmdStart(0x80+column);
239         }
240         else {
241                 LCD_WriteCmdStart(0xB8+column); // IEEE change this value to 0xB
    F+column
242         }
243 }
244
245 enum LI_States { LI_Init1, LI_Init2, LI_Init3, LI_Init4, LI_Init5, LI_Init6,
246         LI_WaitDisplayString, LI_Clr, LI_PositionCursor, LI_DisplayChar, LI_Wait
    Go0 } LI_State;
247
248 void LI_Tick() {
249         static unsigned char i;
250         switch(LI_State) { // Transitions
251                 case −1:
252                         LI_State = LI_Init1;
253                         break;
254                 case LI_Init1:
255                         LI_State = LI_Init2;
256                         i=0;
257                         break;
258                 case LI_Init2:
259                         if (i<10) { // Wait 100 ms after power up
260                                 LI_State = LI_Init2;
261                         }
262                         else {
263                                 LI_State = LI_Init3;
264                         }
265                         break;
266                 case LI_Init3:
267                         LI_State = LI_Init4;
268                         LCD_WriteCmdEnd();
269                         break;
270                 case LI_Init4:
271                         LI_State = LI_Init5;
272                         LCD_WriteCmdEnd();
273                         break;
274                 case LI_Init5:
```

```
                        LI_State = LI_Init6;
                        LCD_WriteCmdEnd();
                        break;
                case LI_Init6:
                        LI_State = LI_WaitDisplayString;
                        LCD_WriteCmdEnd();
                        break;
                ////////////////////////////////////////
                case LI_WaitDisplayString:
                        if (!LCD_go_g) {
                                LI_State = LI_WaitDisplayString;
                        }
                        else if (LCD_go_g) {
                         LCD_rdy_g = 0;
                                LI_State = LI_Clr;
                        }
                        break;
                case LI_Clr:
                        LI_State = LI_PositionCursor;
                        LCD_WriteCmdEnd();
                        i=0;
                        break;
                case LI_PositionCursor:
                        LI_State = LI_DisplayChar;
                        LCD_WriteCmdEnd();
                        break;
                case LI_DisplayChar:
                        if (i<16) {
                                LI_State = LI_PositionCursor;
                                LCD_WriteDataEnd();
                        i++;
                        }
                        else {
                                LI_State = LI_WaitGo0;
                                LCD_WriteDataEnd();
                        }
                        break;
                case LI_WaitGo0:
                        if (!LCD_go_g) {
                                LI_State = LI_WaitDisplayString;
                        }
                        else if (LCD_go_g) {
                                LI_State = LI_WaitGo0;
                        }
                        break;
                default:
                        LI_State = LI_Init1;
                } // Transitions

        switch(LI_State) { // State actions
                case LI_Init1:
                 LCD_rdy_g = 0;
                        break;
                case LI_Init2:
                        i++; // Waiting after power up
                        break;
                case LI_Init3:
                        LCD_WriteCmdStart(0x38);
                        break;
                case LI_Init4:
                        LCD_WriteCmdStart(0x06);
                        break;
                case LI_Init5:
                        LCD_WriteCmdStart(0x0F);
                        break;
                case LI_Init6:
                        LCD_WriteCmdStart(0x01); // Clear
                        break;
                ////////////////////////////////////////
                case LI_WaitDisplayString:
                        LCD_rdy_g = 1;
                        break;
                case LI_Clr:
```



```
                        LCD_WriteCmdStart(0x01);
                        break;
                case LI_PositionCursor:
                        LCD_Cursor(i);
                        break;
                case LI_DisplayChar:
                        LCD_WriteDataStart(LCD_string_g[i]);
                        break;
                case LI_WaitGo0:
                        break;
                default:
                        break;
        } // State actions
}
//--------END LCD interface synchSM-----------------------------------------
---


// SynchSM for testing the LCD interface -- waits for button press, fills LCD wi
th repeated random num

enum LT_States { LT_s0, LT_WaitLcdRdy, LT_WaitButton, LT_FillAndDispString,
LT_HoldGo1, LT_WaitBtnRelease } LT_State;

void LT_Tick() {
        static unsigned char i, x, c;
        switch(LT_State) { // Transitions
                case -1:
                        LT_State = LT_s0;
                        break;
                case LT_s0:
                        LT_State = LT_WaitLcdRdy;
                        break;
                case LT_WaitLcdRdy:
                        if (!LCD_rdy_g) {
                                LT_State = LT_WaitLcdRdy;
                        }
                        else if (LCD_rdy_g) {
                                LT_State = LT_WaitButton;
                        }
                        break;
                case LT_WaitButton:
                        if (keypad_number == prev_key_num) {
                                LT_State = LT_WaitButton;
                        }
                        else if (!(keypad_number == prev_key_num)) {
            // Button active low
                                LT_State = LT_FillAndDispString;
                        }
                        break;
                case LT_FillAndDispString:
                        LT_State = LT_HoldGo1;
                        break;
                case LT_HoldGo1:
                        LCD_go_g=0;
                        LT_State = LT_WaitBtnRelease;
                        break;
                case LT_WaitBtnRelease:
                        if ((keypad_number == prev_key_num)) {
             // Wait for button release
                                LT_State = LT_WaitBtnRelease;
                        }
                        else if (!(keypad_number == prev_key_num)) {
                                LT_State = LT_WaitLcdRdy;
                        }
                        break;
                default:
                        LT_State = LT_s0;
                } // Transitions

        switch(LT_State) { // State actions
                case LT_s0:
                        LCD_go_g=0;
```

```
                        strcpy(LCD_string_g, "1234567890123456"); // Init, but nev
er seen, shows use of strcpy though
                        break;
                case LT_WaitLcdRdy:
                        break;
                case LT_WaitButton:
                        break;
                case LT_FillAndDispString:
                        //x = rand() % 10; // x is rand num 0-9
                        x = keypad_number;
         c = (char)(/*((int)'0')+*/x); // c is ascii of num x
                        for (i=0; i<16; i++) { // Fill string with c
                                LCD_string_g[i] = c;
         }
                        LCD_string_g[i] = '\0'; // End-of-string char
                        LCD_go_g = 1; // Display string
                        break;
                case LT_HoldGo1:
                        break;
                case LT_WaitBtnRelease:
                        break;
                default:
                        break;
        } // State actions
}

/*****************************************************************************/
/* Perform all of the initialization operations here.                        */
/*****************************************************************************/
void Init_LCD_Task()
{
        DDRB = 0xFF; // Set port B to output
        DDRD = 0xFF; // Set port D to output
        DDRA = 0xFC; // PA0 is an input
        LI_State = -1;
        LT_State = -1;
}

/*****************************************************************************/
/* Process_LCD_Task()                                                        */
/* This method will advance the LCD task state machine one state per         */
/* timer tick.                                                               */
/*****************************************************************************/
int Process_LCD_Task(int state)
{
        LI_Tick();
        LT_Tick();

        // Don't really needed to pass state information to the scheduler
        return 1;
}

////////////////////////////////////////////////////////////////////////////////
//Functionality - finds the greatest common divisor of two values
//Parameter: Two long int's to find their GCD
//Returns: GCD else 0
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;
}

////////////////////////////////////////////////////////////////////////////////
//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 period;           //Task period
        unsigned long elapsedTime;      //Time elapsed since last task tick
        int (*TickFct)(int);            //Task tick function
} task;

// Implement scheduler code from PES.
void RunTaskScheduler()
{
        // Initialize the tasks.
        Init_KeyPad_Task();
        Init_LCD_Task();

        unsigned long int LCD_Tick    = 10;
        unsigned long int KeyPad_Tick = 10;

        // Calculate the GCD of the LCD and KeyPad tasks
        unsigned long int tmpGCD = findGCD(LCD_Tick, KeyPad_Tick);

        //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 LCD_period    = LCD_Tick/GCD;
        unsigned long int KeyPad_period = KeyPad_Tick/GCD;

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

        // Task 1
        task1.state = -1;//Task initial state.
        task1.period = LCD_period;//Task Period.
        task1.elapsedTime = LCD_period;//Task current elapsed time.
        task1.TickFct = &Process_LCD_Task;//Function pointer for the tick.

        // Task 2
        task2.state = -1;//Task initial state.
        task2.period = KeyPad_period;//Task Period.
        task2.elapsedTime = KeyPad_period;//Task current elapsed time.
        task2.TickFct = &Process_KeyPad_Task;//Function pointer for the tick.

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

        // Run all tasks forever....
        unsigned short i; // Scheduler for-loop iterator
        while(1)
        {
                // Scheduler code
                for ( i = 0; i < 2; 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]->st
ate);
                                // Reset the elapsed time for next tick.
                                tasks[i]->elapsedTime = 0;
                        }
                        tasks[i]->elapsedTime += 1;
                }

                // Process_LCD_Task(1);
                while(!TimerFlag);
                TimerFlag = 0;
        }
}
```

```
562
563
564 int main(void)
565 {
566         // The Task scheduler does all of the real work
567         RunTaskScheduler();
568 }
```