Lab 7 Three Story Elevator

by

Taylor Hancuff

Choong Huh

collaboration with the group of

Brett Carter

David Parrot

CS466 Embedded Systems

LABORATORY REPORT

Computer Science

Washington State University

2014-MAY-7

Objectives

The goal of this laboratory project is to design an operating system for a master board and a slave board to control a 4-motor elevator system. Software for the elevator control is written in dynamic C and runs on BL4S100. Inter-board communication enables each Rabbit board to control two motors and synchronize a simultaneous elevator motion.

In addition to achieving communication between two boards, the elevator will simulate three test modes.

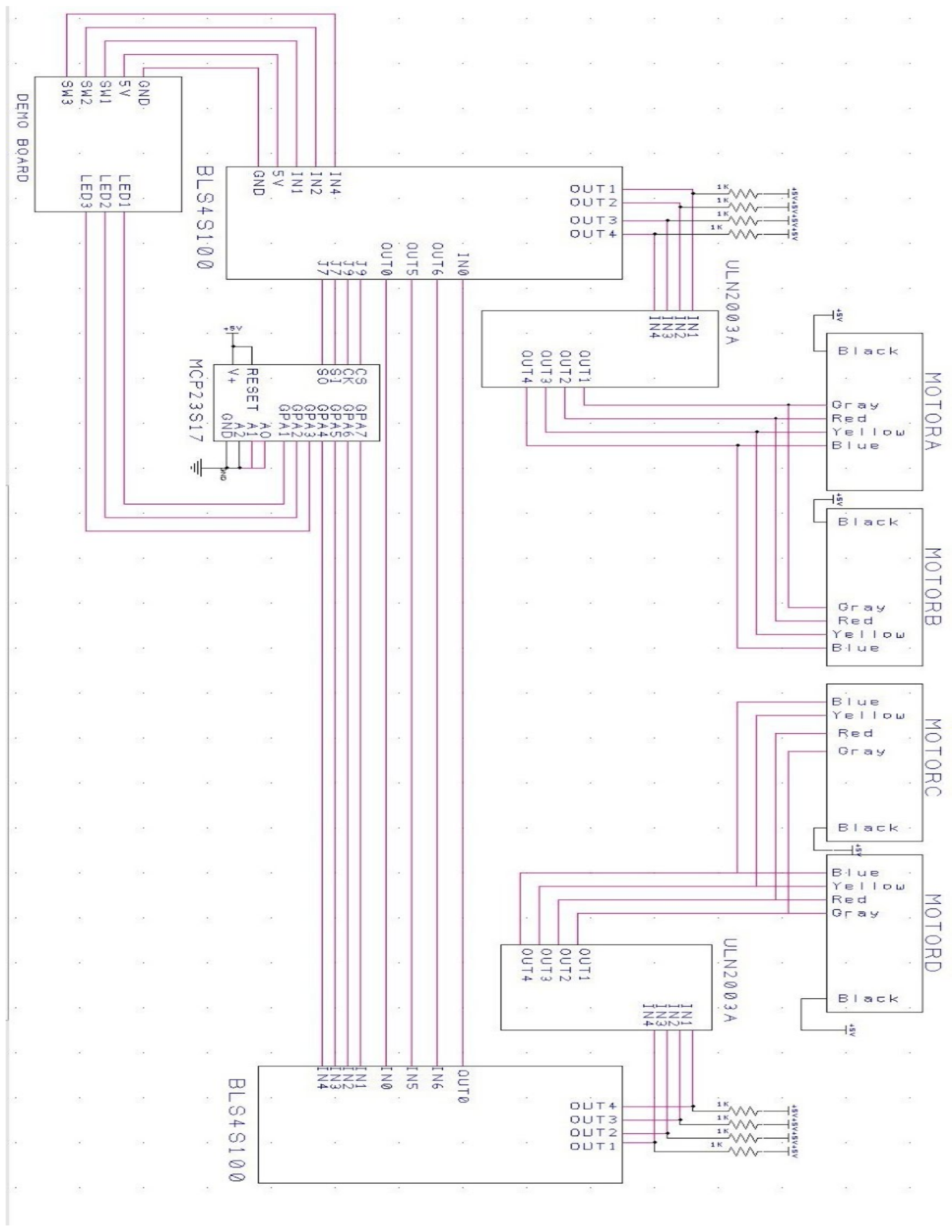
mode 1. To move the elevator up and down at least 3 inches for two minutes.
mode 2. To move the elevator up and down 10 times while calculating prime numbers
mode 3. To simulate elevator control by going up and down upon button press

Apparatus

Materials:

- BL4S100 Rabbit Board (2)
- ULN2003A darlington array (2)
- MCP23S17 GPIO expanders (2)
- Demo Board
- Unipolar Stepper Motors (4)
- Power supply cable (2)
- Usb Programming Cable (2)
- Power Generator 7.5v 2.8A (1)
- Oscilloscope
- Digital Multimeter
- Probing Wires
- Dynamic C
- solid core wires
- Flat Blade screwdriver

The motors are controlled by four outputs on each of the BL4S100 boards.. They are sequenced following the data sheet for the unipolar stepper motors in the software. The pullup resistors show that they will be pulled up to 5v, but in lab we ended up using the 7.5v standard instead. The ULN2003A darlington array is used as a safety precaution for the rabbit boards. The GPIO expanders are used as a solution for the I/O constraints. The schematic is as Follows:



Methods

Two separate program files (Elevator.c, elevator_slave.c) were created to each control the master and the slave board. Master control program creates and runs three tasks that govern the control of the three elevator modes: **mode1**, **mode2**, **mode3.** Mode 1 will run the elevator up and down for two minutes, mode 2 will go up and down ten times, stopping for a half second at each floor while calculating primes in a different task., mode 3 gives the user the power to move the elevator up or down while the led’s light displaying which floor you are currently on. Each task is given its interrupt routines which initially wait to grab a semaphore before being allowed to run. Switch 1 on the demo board is mode1, switch two is mode two, and switch three is mode three. While in Mode three the user can use button 1 to move down and button 2 to move up. Since the master board is connected to a demo board, and three buttons are set up to send signals to the master board's in1, in2, and in4 we have Handler functions added to the aforementioned input channels, so that the semaphore becomes available and the ISR of each modes can be run when the button is pressed. With each of the led's tied to a certain floor that the elevator is on, we needed extra i/o, the gpio expander bits [3:1] are assigned to perform this connection.

For programming ease, **motorcontroller( )** takes angular degree integer as an argument, converts it to the number of full-stepping required to rotate the motor accordingly, and calls digOut to pulse the lines for the two motors. **motorcontroller( )** also sends the identical motor control sequence to the slave board via **GPIO(fullSeq[row]);** call. fullSeq[ ][ ] is an array of 4-int arrays that simulate a full-step motor rotation when **row** gets incremented and all four control sequences get sent to the motors via digOut( ). With the **GPIO()** function we are encapsulating the SPI interface via a bit bang protocol. We first call zwritebyte(). GPIO() handles a four bit array that passes the motor control sequence from the global variable to the gpio ports GPIO7-GPIO4. led_GPIO() handles all the led pulses, it will pass which floor, inside floor controller where call happens, to GPIO3-GPIO1. It acts the same as the GPIO() except for which bits are set. Since when the slave board is reading the motor controls the led is blank we don't have to worry about what these signals are when we set the led therefore the upper bits are don't cares at this point in the program. The transfer function that these are both calling is native to the start up code from angel. We initialize the rabbit board so we can use the ports on J9 and J7 as generalized IO. We use them as the CS, CK, MISO, MOSI lines. When the transfer function is called it immediately drops CS low, it then pulses out each bit of data corresponding to the clock. As this happens it is also storing any bytes of data coming back from the slack on the miso line. This is in case we ever need to read back from it. Although since we are only outputting the only time we needed to actually read from the device was when we were initially setting the device up and needed to verify the chip was on by reading registers inside it.

After writing to the GPIO, master pulses out0 pin, which reaches the slave's in0 interrupt to signal the beginning of data transfer. Slave board, in return, sends "**ack**" signal to the master's in0 to tell the master that it is ready for the input. Master sees that the slave is ready, sends a "**go**" signal to the slave, and controls its motors. Respectively, the slave controls its motors with the command received from GPIO upon receiving the "go" signal from the master. Since we are only sending four bits at a time we are able to concurrently run the motor as we read in from the gpio. The Primes function acts only in mode two. The in6 on the slave board is set as an interrupt to calculate only when this line goes high. Our slave code has to be recompiled with the primes task active in order for this mode to work. When we want modes 1 and 3 the primes task must be commented out.

Slave board control program creates and runs three tasks: INQueue**,** DEQueue**,** PRimes. INQueue accepts the master's data from GPIO and is triggered when in0 goes high, deQueue outputs the received data to the motors by pulling from the array that was initially set by INQueue, and primes calculates prime numbers for the master board's mode2 to ease its computing load. Due to issues with The darlington array driver going haywire we had some minor setbacks with our timing as far as being able to set the slave board as master and vice versa. We ended up rewiring two whole new boards as well as all new components to get our project fully function by monday of finals week.

Data

No data recording was done in this experiment other than the observation of elevator motion.
Links to the demonstration are Below:

Mode1:
    http://youtu.be/qs8pE7vVwTo
Mode 2:
    http://youtu.be/iBWSOMzbfL0
Mode 3:
    http://youtu.be/P7N3Fkiuubw

From mode 2 we were able to obtain 37 prime numbers.
The power use of our system while the motors were turning was 7.5v * 2.8A = 21 Watts.

Results and Analysis

The master / slave program successfully demonstrated the three elevator operation modes. The optimal elevator speed was chosen for each modes to maintain the synchronization while not being too slow.

Conclusion

The experiment's goal was to design an operating system for a master board and a slave board to control a 4-motor elevator system. Master-slave communication was achieved via SPI and GPIO expander. RTOS task system was used to allow modes to keep listening for button input, but regulated with semaphores to not continuously check and waste computing resources. Three modes in the specifications have successfully been implemented to the system and demonstrated.

This experiment presented us with opportunities to familiarize with elements of embedded systems covered in the lectures and previous lab activities. We believe that completion of this project was a good learning experience and a preparation for potential professional occasions in the future that involves embedded systems.

Code:

```
/*
Elevator Master code.

David Parrott, Brett Carter, Choong Huh, Taylor Hancuff
CS 466 Fall 2014
Final Project
*/
#class auto
#define OS_MAX_EVENTS              4              // Maximum number of events (semaphores,
                                                  // queues, mailboxes)
#define OS_MAX_TASKS               11             // Maximum number of tasks system can
                                                  // create (less stat and idle tasks)
#define OS_TASK_CREATE_EN          1              // Enable normal task creation
#define OS_TASK_CREATE_EXT_EN1             // Enable extended task creation
#define OS_TASK_STAT_EN            1              // Enable statistics task creation
#define OS_MBOX_EN                 1              // Enable mailboxes
#define OS_MBOX_POST_EN            1              // Enable MboxPost
#define OS_TIME_DLY_HMSM_EN        1              // Enable OSTimeDlyHMSM
#define STACK_CNT_512       8              // number of 512 byte stacks (application
                                                  // tasks + stat task + prog stack)
#define OS_MAX_MEM_PART            10             // Maximum number of memory partions in
                                                  // system
#define OS_MEM_EN                  1              // Enable memory manager
#define OS_Q_EN                    1              // Enable queues
#define OS_TICKS_PER_SEC           64
#define TASK_STK_SIZE              512            // Size of each task's stacks (# of
                                                  // bytes)
#define TASK_START_ID              0              // Application tasks IDs
#define TASK_START_PRIO            10             // Application tasks priorities
#define RSB_MAX_ISR                6
#define V0                         ((void*) 0)
#define LOW                        (0)
#define HIGH                       (1)
#define zSetCS(x)                  WrPortI(PDB6R, V0, (x)?(1<<6):0)
#define zSetSCK(x)                 WrPortI(PDB7R, V0, (x)?(1<<7):0)
#define zSetMOSI(x)                WrPortI(PDB3R, V0, (x)?(1<<3):0)
#define zGetMISO()                 ((RdPortI(PEDR)&(1<<6))?1:0)

#use "BL4S1xx.LIB"
#use "ucos2.lib"
#use "rand.lib"
#use "rtclock.lib"
#use "costate.lib"
#include <time.h>

typedef char uint8_t;

/*
Declarations of global ints to handle interrupts on button presses
```

```
*/
int button_press1;
int button_press2;
int button_press3;
int up_flag, down_flag, exit_flag;

int bp1_handle;
int bp2_handle;
int bp3_handle;

int bp31_handle;
int bp32_handle;
int bp33_handle;

OS_EVENT *btn1sem, *btn2sem, *btn3sem;

/*
Function prototypes.
*/
void           floorController(int desiredFloor);
void           motorController(int degrees);
void           slaveController(int degrees);
uint8_t        transfer(uint8_t out);
uint8_t        zReadByte(uint8_t address);
void           zWriteByte(uint8_t address, uint8_t data);
void           test2Mins();
void           floors();
void           GPIO(int* array);

/*
These arrays hold the sequences for half and full stepping
*/
int fullSeq[][4] = {{1,0,0,0},{0,0,1,0},{0,1,0,0},{0,0,0,1}};
int halfSeq[][4] = {{1, 0, 0, 0}, {1, 0, 1, 0}, {0, 0, 1, 0}, {0, 1, 1, 0}, {0, 1, 0, 0}, {0,
1, 0, 1}, {0, 0, 0, 1}, {1, 0, 0, 1}};

/*
Global int keeps track of what floor the elevator is currently on.
*/
int currentFloor = 1;

/*
The next several functions were provided by Miller Lowe

function to initialize the pins.
*/
void pinInit(){
       int mask;
       WrPortI(PDFR, V0, RdPortI(PDFR) & ~((1<<3) | (1<<6) | (1<<7)));
  WrPortI(PDDCR, V0, RdPortI(PDDCR) & ~((1<<3) | (1<<6) | (1<<7)));
  WrPortI(PDCR, V0, 0);
  WrPortI(PDDDR, V0, RdPortI(PDDDR) | ((1<<1) | (1<<3) | (1<<6) | (1<<7)));

  WrPortI(PEFR, V0, RdPortI(PEFR) & ~((1<<6)));
  mask = RdPortI(PEDDR);
  mask &= ~(1<<6);
  mask |= (1<<3);
  WrPortI(PEDDR, V0, mask);
  WrPortI(PEDCR, V0, RdPortI(PEDCR) & ~(1<<3));

}

/*
Send the buffer sent as o'ut' to the pins
*/
uint8_t transfer(uint8_t out){
       uint8_t i;
   uint8_t in = 0;
   zSetSCK(LOW);
   for(i = 0; i<8; i++){
       in <<= 1;
      zSetMOSI(out & 0x80);
      zSetSCK(HIGH);
      in += zGetMISO();
      zSetSCK(LOW);
      out <<= 1;
   }
   zSetMOSI(0);
   return(in);
}

/*
read from address: 0x12
*/
uint8_t zReadByte(uint8_t address){
       uint8_t preRead = 0x41;
   uint8_t value;
   zSetCS(LOW);
   transfer(preRead);
   transfer(address);
   value = transfer(0);
   zSetCS(HIGH);
   return value;
}

/*
write to address: 0x13
*/
void zWriteByte(uint8_t address, uint8_t data){
       uint8_t preWrite = 0x40;
   zSetCS(LOW);
   transfer(preWrite);
   transfer(address);
   transfer(data);
   zSetCS(HIGH);

}

/*
Set output pins on the GPIO to correspond to 1's and 0's passed in array.

Assumptions:
The relevant information is contained in the first for indices of arrary[]
and that those indices contain either a '1' or a '0'
*/
void GPIO(int* array)
{
   uint8_t Byte;
   Byte =
array[0]*(1<<7)|array[1]*(1<<6)|array[2]*(1<<5)|array[3]*(1<<4)|(1<<3)|(1<<2)|(1<<1)|(1<<0);
   zWriteByte(0x00,0x00);
   zWriteByte(0x12,Byte);
}

/*
Set pins leading to the LEDs from the GPIO
*/
void led_GPIO(int* array)
{
       uint8_t Byte;
   Byte = array[0]*(1<<3)|array[1]*(1<<2)|array[2]*(1<<1);
   zWriteByte(0x00,0x00);
   zWriteByte(0x12,Byte);
}

/*
Calculates the number of degrees required to move to the selected floor
Checks to ensure that floor is within the range the elevatoris capable
of moving to. Passes the calculated number of degrees into motorController()
where the actual setting of outputs is done
*/
void floorController(int desiredFloor){
   int netChange;
   int degrees;
   int i;
   int ledarray[4];

       if(desiredFloor <= 0 || desiredFloor > 4){
       return;
   }
   netChange = currentFloor - desiredFloor;
   if(netChange != 0){
       degrees = (netChange * 360);
      motorController(degrees);
              currentFloor = desiredFloor;
      for(i = 1; i < 5; i++){
       if(i == currentFloor){
              ledarray[i-1] = 0;
            //printf("led %i is on \n",i-1);
        }else{
              ledarray[i-1] = 1;
            //printf("led %i is off \n",i-1);
        }
       }
        led_GPIO(ledarray);
   }else{
       return;
   }
}

/*
Takes a number of degrees and converts it to full steps. That number
is used to step through an array containing the correct sequence for
a unipolar stepper motor. Degrees can be any int, positive or negative,
so that the elevator can rotate up or down.

Before turning the motors controlled by the Master board, GPIO() is called
to set the output pins into a sequence that is read by the Slave. The 'read'
line is pulsed with a 1ms delay to ensure it gets read. When an
acknowledgement is received the 'go' line is pulsed before Master begins
turning motors.

negative is clock-wise
positive is counter clock-wise
*/
void motorController(int degrees){
   int i;
   int row;
   int col;
   float steps = degrees / (4*7.5);
   float temp;
   if(steps < 0){
      temp = fabs(steps);
      printf("steps: %lf temp: %lf\n",steps, temp);
      for(i = 0; i < (temp-1); i++){
         for(row = 3; row >= 0; row--){
              GPIO(fullSeq[row]);
              digOut(0,1);
              OSTimeDlyHMSM(0,0,0,1);
              digOut(0,0);
```

```
               while(digIn(0) == 0){
               }
               digOut(5,1);
               OSTimeDlyHMSM(0,0,0,1);
               digOut(5,0);
               for(col = 3; col >= 0; col--){
               digOut(col + 1, fullSeq[row][col]^0x1);
                      }
             OSTimeDlyHMSM(0,0,0,9);
       }
      }
   }else{
       for(i = 0; i < (steps-1); i++){
                      for(row = 0; row < 4; row++){
                             GPIO(fullSeq[row]);
              digOut(0,1);
           OSTimeDlyHMSM(0,0,0,1);
                      digOut(0,0);
               while(digIn(0) == 0){
                      }
               digOut(5,1);
               OSTimeDlyHMSM(0,0,0,1);
               digOut(5,0);
                      for(col = 0; col < 4; col++){
                      digOut(col + 1, fullSeq[row][col]^0x1);
                      }
                      OSTimeDlyHMSM(0,0,0,9);
               }
       }
   }
}

/*
Tracks timing in mode 1

Creates 2 time_t structs and stores their difference in result every 12 seconds.
*/
void mode1_helper(){
  int i;
  double result;
  time_t time1, time2;
      time(&time1);
      time(&time2);
      result = difftime(time2, time1);
      while(result < 12){
        floorController(3);
        floorController(1);
          time(&time2);
          result = difftime(time2, time1);
          printf("%lf\n", result);
      }
}

/*
Handlers for ISRs
*/
root void mode1_handle(){
       OSSemPost(btn1sem);
   RSB_CLEAR_ALL_IRQ(bp1_handle);
}

root void mode2_handle(){
 OSSemPost(btn2sem);
 RSB_CLEAR_ALL_IRQ(bp2_handle);
}

root void mode3_handle(){
 OSSemPost(btn3sem);
 RSB_CLEAR_ALL_IRQ(bp3_handle);
}

/*
Handlers for mode 3 button presses
*/
root void m3h1(){
       up_flag = 1;
   RSB_CLEAR_ALL_IRQ(bp31_handle);
}

root void m3h2(){
       down_flag = 1;
   RSB_CLEAR_ALL_IRQ(bp32_handle);
}

root void m3h3(){
       exit_flag = 1;
   RSB_CLEAR_ALL_IRQ(bp33_handle);
}

/*
Governs behavior in mode 1

calls mode1_helper() 20 times. The handler lasts for 12 seconds
resulting in a total exdcution time of 120 seconds
*/
void mode1(){
  int i;
  printf("mode 1\n");

  while(1){
```

```c
    OSSemPend(btn1sem, 0, NULL);
    for(i = 0; i < 10; i++){
       mode1_helper();
      printf("i = %d\n",i);
    }
  }
}

/*
Governs behavior for mode 2

Traverses all the floors with a 500ms delay at each floor
*/
void mode2(){
 int i;
 puts("mode2\n");
       while(1){
    OSSemPend(btn2sem, 0, NULL);
    for(i = 0; i < 10; i++){
      floorController(2);
      OSTimeDlyHMSM(0,0,0,500);
      floorController(3);
      OSTimeDlyHMSM(0,0,0,500);
      floorController(2);
      OSTimeDlyHMSM(0,0,0,500);
      floorController(1);
      OSTimeDlyHMSM(0,0,0,500);
    }
    digOut(6,1);
    OSTimeDlyHMSM(0,0,0,1);
    digOut(6,0);
       }
}

/*
Governs behavior for mode 3

Sets up new ISRs to respond to button presses before dropping into
a while loop that will allow navigation between floors
*/
void mode3(){
       printf("mode 3\n");
       while(1){
    OSSemPend(btn3sem, 0, NULL);
    enableISR(bp1_handle, 0);
    enableISR(bp2_handle, 0);
    enableISR(bp3_handle, 0);
    bp31_handle = addISRIn(1, 0, &m3h1);
    bp32_handle = addISRIn(2, 0, &m3h2);
    bp33_handle = addISRIn(4, 0, &m3h3);
    setExtInterrupt(1, BL_IRQ_FALL, bp31_handle);
    setExtInterrupt(2, BL_IRQ_FALL, bp32_handle);
       setExtInterrupt(4, BL_IRQ_FALL, bp33_handle);
    enableISR(bp31_handle, 1);
       enableISR(bp32_handle, 1);
       enableISR(bp33_handle, 1);

       /**
         code to handle mode 3 operations
       **/

//====================================================
// button 3 will exit out of this mode, so
// hopefully while(button_press3 != 1) will let it break out
// of the loop when button 3 is pressed.
// button 1 is down and button 2 is up.

    while(1){
      if(exit_flag == 1){
       exit_flag = 0;
       break;
      }
      if(up_flag == 1){
       puts("going up!");
       floorController(currentFloor+1);
        up_flag = 0;
      }
      if(down_flag == 1){
       puts("going down!");
       floorController(currentFloor-1);
        down_flag = 0;
      }
    }

//=======RESET BUTTON_PRESS================

    enableISR(bp31_handle, 0);
    enableISR(bp32_handle, 0);
    enableISR(bp33_handle, 0);

       setExtInterrupt(1, BL_IRQ_FALL, bp1_handle);
       setExtInterrupt(2, BL_IRQ_FALL, bp2_handle);
       setExtInterrupt(4, BL_IRQ_FALL, bp3_handle);
       enableISR(bp1_handle, 1);
       enableISR(bp2_handle, 1);
       enableISR(bp3_handle, 1);
  }
}
```

```
/*
Set up and launch tasks that will listen for button presses
*/
void main(){
       int initarray[3] = {1,1,1};
  btn1sem = OSSemCreate(0);
  btn2sem = OSSemCreate(0);
  btn3sem = OSSemCreate(0);

  OSInit();
  brdInit();
  pinInit();

  //button press interupts set-up//
  bp1_handle = addISRIn(1, 0, &mode1_handle);  //calls mode 1 function on input0 press
  bp2_handle = addISRIn(2, 0, &mode2_handle);  //calls mode 2 function on input1 press
  bp3_handle = addISRIn(4, 0, &mode3_handle);  //calls mode 3 function on input2 press

  //initialize interrupt line to zero
  digOut(0,0);
  digOut(5,0);

 //LED GPIO set to zero
 led_GPIO(initarray);

  setExtInterrupt(1, BL_IRQ_FALL, bp1_handle);
  setExtInterrupt(2, BL_IRQ_FALL, bp2_handle);
  setExtInterrupt(4, BL_IRQ_FALL, bp3_handle);

  //enable the 3 ISR's.
  enableISR(bp1_handle, 1);
  enableISR(bp2_handle, 1);
  enableISR(bp3_handle, 1);

  //launch mode tasks
  OSTaskCreateExt(mode3,
    (void *)0,
            11,
            2,
            TASK_STK_SIZE,
    (void *)0,
            OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

  OSTaskCreateExt(mode1,
    (void *)0,
            10,
            1,
            TASK_STK_SIZE,
    (void *)0,
            OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
  OSTaskCreateExt(mode2,
    (void*)0,
           12,
           3,
           TASK_STK_SIZE,
    (void*)0,
           OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

  OSStart();
}
```

```
/*
David Parrott, Brett Carter, Choong Huh, Taylor Hancuff
CS 466 elevator controller
slave program
*/
#class auto
#define  OS_MAX_EVENTS             4         // Maximum number of events (semaphores, queues,
mailboxes)
#define OS_MAX_TASKS              11         // Maximum number of tasks system can create (less
stat and idle tasks)
```

```
#define OS_TASK_CREATE_EN       1       // Enable normal task creation
#define OS_TASK_CREATE_EXT_EN    1       // Enable extended task creation
#define OS_TASK_STAT_EN          1       // Enable statistics task creation
#define OS_MBOX_EN             1         // Enable mailboxes
#define OS_MBOX_POST_EN          1         // Enable MboxPost
#define OS_TIME_DLY_HMSM_EN    1         // Enable OSTimeDlyHMSM
#define STACK_CNT_512           8       // number of 512 byte stacks (application tasks + stat
task + prog stack)
#define OS_MAX_MEM_PART          10        // Maximum number of memory partions in system
#define OS_MEM_EN                1       // Enable memory manager
#define OS_Q_EN                1        // Enable queues
#define OS_TICKS_PER_SEC       64
#define TASK_STK_SIZE             512      /* Size of each task's stacks (# of bytes)       */
#define TASK_START_ID           0         /* Application tasks IDs                         */
#define TASK_START_PRIO        10      /* Application tasks priorities */

#use "BL4S1xx.LIB"
#use "ucos2.lib"
#use "rand.lib"

#define V0 ((void*) 0)
#define LOW (0)
#define HIGH (1)
#define zSetCS(x) WrPortI(PDB6R, V0, (x)?(1<<6):0)
#define zSetSCK(x) WrPortI(PDB7R, V0, (x)?(1<<7):0)
#define zSetMOSI(x) WrPortI(PDB3R, V0, (x)?(1<<3):0)
#define zGetMISO() ((RdPortI(PEDR)&(1<<6))?1:0)

int      ISRFlag;
int      ISR_handle;
int      primeArray[1000];
int      primeCount;
int      stop_flag;
void      enQueue();
void      deQueue();
void      primes(int n);
void       primeQ(int n);
void       GPIO(int* array);
void      turnMotor(int entries);
int      array[4];

typedef char uint8_t;
uint8_t  transfer(uint8_t out);
uint8_t  zReadByte(uint8_t address);
void     zWriteByte(uint8_t address, uint8_t data);
 int quehandler;
 int dequehandler;
 int primehandler;
 OS_EVENT *reading, *DQ, *PR;
int fullSeq[][4] = {{1,0,0,0},{0,0,1,0},{0,1,0,0},{0,0,0,1}};
```

```
//BEGIN CODE FROM MILLER LOWE
// This is the aggregate data structure that you wish to queue between two tasks
//
typedef struct {
    int d[4];
} mQueueEntry_t;

//
// A Queue handle points to one of these..
typedef struct {
    int elementSize;
    int elementCount;
    int inQueue;
    OS_EVENT * queue;
    OS_MEM * memPool;
} labQueue_t;

// API return Values
typedef enum {
    LABQ_OK = 0,
    LABQ_ERROR = -1,
    LABQ_TIMEOUT = -2,
    LABQ_FULL = -3,
} labQueueReturn_t;

// Number of elements in the queue
#define Q1_ELEMENTS 900

// You must allocate storage for the queue data and funny pointer array
static void * q1_pArray[Q1_ELEMENTS];  // array of pointers for uCos Queue
static char * q1_eStorage[Q1_ELEMENTS * sizeof(mQueueEntry_t)]; // element storage
labQueue_t lq;

//
// -------------------------------- labQueue Implementation
// -------------------------------- Could be in a seperate file or paste into yours..
// Requires OS Defines (Prios to the '#use "ucos2.lib"' statement
// OS_MAX_EVENTS  +1 ber Queue allocated
// OS_MEM_EN      Must be defined/enabled
// OS_Q_EN        Shoudl be 1

//
// labQueueCreate -
//
// Fills out a queue structure that you have to allocate and hand in to the create.
//
labQueueReturn_t labQueueCreate(    labQueue_t *lq, int elementSize, int elementCount, void *
elementStorage, void *pointerArray[]){
    INT8U error;
    lq->inQueue = 0;
    lq->elementSize = elementSize;
    lq->elementCount = elementCount;
    lq->queue = OSQCreate(pointerArray, elementCount);
    assert(lq->queue);
    lq->memPool = OSMemCreate(elementStorage, elementCount, elementSize, &error);
    assert(error==OS_NO_ERR);
    return (LABQ_OK);
}



//
// Give a queue handle and a pointer to the element, copy the data into the
// the queue.  This will fail for the put fails, not block..  We could extend it to
// blocking with a semaphore and a little work.. Who's motovated for that?
//
labQueueReturn_t labQueuePut( labQueue_t *lq, void *element){
    INT8U error;
    void * qElement;
    labQueueReturn_t result;

    qElement = OSMemGet(lq->memPool, &error);

    switch (error)
    {
    case OS_NO_ERR:
         lq->inQueue++;
        memcpy(qElement, element, lq->elementSize);
        error = OSQPost( lq->queue, qElement);
        switch (error)
        {
        case OS_NO_ERR:
            result = LABQ_OK;
            break;
        case OS_Q_FULL:
            error = OSMemPut(lq->memPool, qElement);
            assert( error == OS_NO_ERR);
            result = LABQ_FULL;
            break;
        default:
            error = OSMemPut(lq->memPool, qElement);
            assert( error == OS_NO_ERR);
            result = LABQ_ERROR;
            assert(0);
            break;
        }
        break;
    case OS_MEM_NO_FREE_BLKS:
        result = LABQ_FULL;
        break;
    default:
        result = LABQ_ERROR;
        assert(0);
        break;
    }
    return result;
}

//
// get the oldest element in the queue.  Fills out the element buffer
// that you pass in.
//
labQueueReturn_t labQueueGet( labQueue_t *lq, void * element, int timeout){
    void *qElement;
    INT8U error;
    labQueueReturn_t result;

    assert(timeout <= 0x0000ffff);
    qElement = OSQPend (lq->queue, timeout, &error);
    switch (error)
    {
    case OS_NO_ERR:
        memcpy(element, qElement, lq->elementSize);
        error = OSMemPut(lq->memPool, qElement);
        assert( error == OS_NO_ERR);
        result = LABQ_OK;
        lq->inQueue--;
        break;
    case OS_TIMEOUT:
        result = LABQ_TIMEOUT;
        break;
    default:
        assert(0);
        result = LABQ_ERROR;
        break;
    }
   return result;
}

void pinInit(){
   int mask;
   WrPortI(PDFR, V0, RdPortI(PDFR) & ~((1<<3) | (1<<6) | (1<<7)));
   WrPortI(PDDCR, V0, RdPortI(PDDCR) & ~((1<<3) | (1<<6) | (1<<7)));
   WrPortI(PDCR, V0, 0);
   WrPortI(PDDDR, V0, RdPortI(PDDDR) | ((1<<1) | (1<<3) | (1<<6) | (1<<7)));

   WrPortI(PEFR, V0, RdPortI(PEFR) & ~((1<<6)));
   mask = RdPortI(PEDDR);
   mask &= ~(1<<6);
   mask |= (1<<3);
   WrPortI(PEDDR, V0, mask);
   WrPortI(PEDCR, V0, RdPortI(PEDCR) & ~(1<<3));
}

uint8_t transfer(uint8_t out){
    uint8_t i;
   uint8_t in = 0;
   zSetSCK(LOW);
   for(i = 0; i<8; i++){
       in <<= 1;
      zSetMOSI(out & 0x80);
      zSetSCK(HIGH);
      in += zGetMISO();
      zSetSCK(LOW);
      out <<= 1;
   }
   zSetMOSI(0);
   return(in);
}
//read from address: 0x12
uint8_t zReadByte(uint8_t address){
    uint8_t preRead = 0x41;
   uint8_t value;
   //assert(address < 16);
   zSetCS(LOW);
   transfer(preRead);
   transfer(address);
   value = transfer(0);
   zSetCS(HIGH);
   return value;
}
//write to address: 0x13
void zWriteByte(uint8_t address, uint8_t data){
    uint8_t preWrite = 0x40;
   //printf("%d \n", address);
  // assert(address < 16);
   zSetCS(LOW);
    //OSTimeDly(100);
   //transfer(preWrite | address);
   transfer(preWrite);
   transfer(address);
   transfer(data);
   zSetCS(HIGH);

}

void fillQueue(){
   mQueueEntry_t qEnt;
   labQueueReturn_t result;
   int i;
   for(i = 0;result != LABQ_FULL;i++){
      result = labQueuePut(&lq, &qEnt);
      //printf("%i \n",i);
   }
}


void GPIO(int* array)
{
   uint8_t Byte;
   int i;
   Byte = array[0]*(1<<7)|array[1]*(1<<6)|array[2]*(1<<5)|array[3]*(1<<4);
   zWriteByte(0x00,0x00);
   zWriteByte(0x12,Byte);

}

void enQueue(){
   int i;
   for(i = 1;i < 5; i++){
      array[i-1] = digIn(i);
      //printf("%i ",array[i-1]);
   }
   //printf("\n");
   digOut(0,1);
   OSTimeDlyHMSM(0,0,0,1);
   digOut(0,0);
}

/*
void enQueue(){
   mQueueEntry_t qEnt;
   int i;
   //printf("enqueue \n");
   for(i = 1; i < 5; i++){
      qEnt.d[i-1] = digIn(i);
      //printf("%i ",qEnt.d[i-1]);
   }
   //printf("\n");

   labQueuePut(&lq, &qEnt);
   digOut(0,1);
   OSTimeDlyHMSM(0,0,0,1);
   digOut(0,0);
   RSB_CLEAR_ALL_IRQ(quehandler);
}
*/

void deQueueAll(){
   int i;
   for(i = 1; i < 5; i++){
      digOut(i,array[i-1]);
   }
   digOut(0,1);
   OSTimeDlyHMSM(0,0,0,1);
   digOut(0,0);
   //printf("dequeued!\n");
}

/*
void deQueueAll(){
   mQueueEntry_t qEnt;
   int i, t;
   //printf("dequeue\n");
   while(lq.inQueue > 0){
      labQueueGet(&lq, &qEnt, 0x0000ffff);
      for(i = 1; i < 5; i++){
         t = qEnt.d[i];
         digOut(i, t);
           OSTimeDlyHMSM(0,0,0,9);
      }
   }
RSB_CLEAR_ALL_IRQ(dequehandler);
}
*/
void primes(int n){
   int i, count, c;
   printf("in primes\n");
   i = 3;
   for(count = 2; count <= n;){
      for(c = 2; c <= i - 1; c++){
         if(i % c == 0)
            break;
      }
      if(c == i){
         primeArray[primeCount] = i;
         primeCount++;
         count++;
      }
      i++;
   }
}

void primeQ(int n){
   int i,j, count, c;
   mQueueEntry_t qEnt;
   OSTimeDlyHMSM(0,0,0,9);
   i = 3;
   printf("in prime function\n");
   for(count = 2; count <= n;){
      OSTimeDlyHMSM(0,0,0,9);
      if(primeCount == 1000){primeCount = 0;}
      for(c = 2; c <= i - 1; c++){
         if(stop_flag == 1){
            for( j = 0; j < primeCount; j++){
             printf("%i\n",primeArray[j]);
            }
            printf("number of primes: %i\n", primeCount);
            stop_flag = 0;
            break;
         }
         OSTimeDlyHMSM(0,0,0,9);
         if(i % c == 0)
            break;
      }
      if(c == i){
         primeArray[primeCount] = i;
         primeCount++;
         count++;
         OSTimeDlyHMSM(0,0,0,9);
         if(stop_flag == 1){
            for( j = 0; j < primeCount; j++){
             printf("%i\n",primeArray[j]);
            }
            printf("number of primes: %i\n", primeCount);
            stop_flag = 0;
            break;
         }
      }
      i++;
   }
}

root void enQueue_handle(){
    OSSemPost(reading);
   OSSemPost(PR);
   RSB_CLEAR_ALL_IRQ(quehandler);
}

root void deQueueAll_handle(){
    OSSemPost(DQ);
   RSB_CLEAR_ALL_IRQ(dequehandler);
}

root void primehandle(){
    stop_flag = 1;
   RSB_CLEAR_ALL_IRQ(primehandler);
}

void INQueue(){
    while(1){
     OSSemPend(reading,0,NULL);
     //printf("queueing\n");
     enQueue();
    }
}

void DEQueue(){
    while(1){
    OSSemPend(DQ, 0,NULL);
    //printf("DEQueue called semaphore grabbed\n");
    deQueueAll();
   }
}

void PRimes(){
   while(1){
      printf("in prime task\n");
      OSSemPend(PR, 0, NULL);
      printf("got semaphore\n");
      primeQ(1000);
   }
}

void main(){

   labQueueReturn_t qr;
   mQueueEntry_t qEnt;
   mQueueEntry_t qE2;
   int i, row, col;
   int num[4];
   reading = OSSemCreate(0);
   DQ = OSSemCreate(0);
   PR = OSSemCreate(0);
   stop_flag = 0;
   OSInit();
   brdInit();
   pinInit();
   digOut(0,0);
   puts("okay");
   primeCount = 0;
   quehandler = addISRIn(0,0,&enQueue_handle);
   dequehandler = addISRIn(5,0,&deQueueAll_handle);
   primehandler = addISRIn(6,0,&primehandle);
   setExtInterrupt(5, BL_IRQ_FALL, dequehandler);
   setExtInterrupt(0, BL_IRQ_FALL, quehandler);
   setExtInterrupt(6, BL_IRQ_FALL, primehandler);
   enableISR(quehandler, 1);
   enableISR(dequehandler, 1);
   enableISR(primehandler, 1);
   qr = labQueueCreate(&lq, sizeof(qEnt.d), Q1_ELEMENTS, q1_eStorage, q1_pArray);

   OSTaskCreateExt(INQueue,
                     (void *)0,
                   10,
                   1,
                   TASK_STK_SIZE,
                   (void *)0,
                   OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

  OSTaskCreateExt(DEQueue,
                     (void *)0,
                   11,
                   2,
                   TASK_STK_SIZE,
                   (void *)0,
                   OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
/*
 OSTaskCreateExt(PRimes,
                     (void *)0,
                   13,
                   3,
                   TASK_STK_SIZE,
                   (void *)0,
                   OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
 */
  OSStart();


}
```