Objective - To raise and lower an elevator in three pre-defined modes using four unipolar stepper motors and two Rabbit BL4S100 control boards. Two motors will be controlled by each board with communication between boards to keep motors turning in synch.

Apparatus - Rabbit Single-Board Computer BL4S100, Demo Board, Assorted wire connections, iMac running Windows 7 and Dynamic C version 10.66, 4 unipolar stepper motors, Darlington Array IC, GPIO Expander, Elevator Apparatus.

Method - When we began the Elevator project we identified three major hurdles: turning the motors in sequence responding to a button press, communication between two Rabbit boards and incorporating interrupts to determine which mode would be activated. We decided to tackle these problems in that order as they seemed to naturally build upon each other.

To turn the motors we wrote two methods: floorController() and motorController(). We chose to encapsulate methods so that we would be able to eliminate redundant checks within the code for each mode. An int representing the desired floor is passed into floorController() which performs checks to ensure the elevator stays in a valid range of floors. It converts the difference between the current floor and the desired floor into a number of degrees and passes that to motorController(). The int passed to motorController() is converted to a number of steps and instructions encoded in a global array are fed first to the GPIO output pins to be read by the Slave board and then directly into the motors controlled by the Master.

The Slave board listens on three direct interrupt lines, “read”, “go” and “primes”. When read is asserted high it records the state of four GPIO pins and stores the values in an array that is used as instructions for the stepper motor. The four index array corresponds to both the GPIO pins and the magnets within the stepper motor. When the pins have been read and encoded an acknowledgement is sent to the Master, which is listening on a direct interrupt line. When Master receives the acknowledgement it pulses the go line starting the motors turning. This process occurs for every step the motor takes. When primes is pulsed high, Slave outputs calculated primes.

The 3 modes were trivial to program using the encapsulated controller methods. A pattern for each and appropriate delays was all that was necessary.

During the final demonstration phase while attempting to implement LED indicators for each floor in mode 3 we encountered a fatal hardware fault. One of the Darlington arrays shorted out damaging the bread board. It took several hours to diagnose this error and required us to completely rewire the entire apparatus. After this rework and some minor fine tuning of code everything worked as intended.

Data - For our mode 2 implementation we were able to have the elevator move to floor 3 and back to floor 1 ten times in 50 seconds. Within this 50 second time period we were able to get our Slave to calculate and print 37 prime numbers.

Analysis - We tried several different methods of communicating and storing instructions between the Master and Slave. Ultimately we determined that reading and executing one instruction at a time was most compatible with how we implemented the elevator controls for each of the modes. We determined that a 9ms wait was necessary between each step of the motor which was more than enough time to set and read a new instruction from the GPIO to the array stored by Slave. While we initially intended to use a queue structure to first communicate all instructions in a series of moves we found that the queue required too much overhead and resulted in diminished speed and synchronization.

When considering synchronization challenges we realized that by using the read and acknowledge lines the possibility for the two boards to get out of synch was eliminated. Master will not advance to the next instruction until Slave has acknowledged receipt. This method did not allow us to ever really contend for the fastest result with mode 2 but it was much more robust.

Conclusion - During the course of this project we experimented with several design approaches for each of the primary components. While some decisions (encapsulating controller methods, using the GPIO for communication) were easy to decide on, others (queue VS read/acknowledge) took experimentation to determine what the most efficient and robust solution would be. Having team members with different areas of expertise was vital to the success of the project. While our final design deviates somewhat from the original specification in that there are obviously predetermined Master and Slave roles, we felt that this was an effective solution that more closely modeled a real world solution.

```
/*
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){
    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();
}

/*
Elevator Slave 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){
    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();
}
```