Project Report: “Rescue Princess Stanford”

ME:218A: Smart Product Design

Zune Lee & Chris Korman
{zune, ckorman}@stanford.edu

December 3, 2004

INDEX

ABSTRACT
DESCRIPTIONS
DIMENSIONED SKETCHES
GEMS OF WISDOM
BILL OF MATERIALS
SCHEMATICS
PSEUDO CODE
SOURCE CODE
This project aims at involving the user in playing a highly interactive and fun game, while letting the developers achieve a meaningful collaboration to apply the principles and techniques learned in lecture and laboratory exercises.

The overall objective of this game is to have the player rescue Princess, Stanford from the severe confinement of Hell, aka “Stanford”, to win the game. In doing so, they will be presented with three different obstacles:

1. Avoiding a randomly timed vane driven via DC motor.
2. Solving a 4-bit binary addition game
3. Finally, catapulting the Princess out of Stanford.

In implementing this game, we used several sensors (IR-LEDs and Photo Transistors), as well as DC motors, red LEDs, 5833 Serial to Parallel Driver, a speaker, and Motorola M68HC32 micro-controller.
DESCRIPTIONS

Overall Structure:

We built a two-tiered structure composed out of foam core and styrofoam. The lower tier is the foundation with its styrofoam supports and failure bin. The upper level contains the success exit, robot arm, catapult, ramp with propeller vane and finally the Binary Math LED array and its associated momentary push buttons.

Input Part:

When a user inserts the “Princess” token into the slot (see input, Figure: overview), a combination IR LED and Photo transistor based circuit detects the tokens presence and simultaneously turns on a green “ready” LED and starts internally a 1 minute count down timer.

Success and Failure Exit:

To detect if the princess passes through the success exit or failure exit, we used IR-LEDs, Photo-transistors, and thin paper doors. When the princess falls down to the exit and hits the door, the IR-LED and transistor would capture the door’s swing outward from the token. Testing indicated that we needed to place the sensor far enough away from the door as to not get false triggering from the device vibrating. Thus, we found a suitable detection zone to indicate that a token actually did pass through. (see Success Exit & Failure Exit, Figure: overview)

1st Feat: The Vane

Upon insertion of the princess into the Stanford contraption, she will be held against a gate. Princess Stanford will need to be safely guided through a vane (see vane, Figure: overview). After the “Princess” (token) has been inserted into the slot, the user can start the game by pressing a ‘Start’ button located on a panel (see panel, Figure: overview) in front of this game that controls the gate. The user must correctly time the release of the princess from the gate to avoid harming the princess by the vane. If the player fails to guide her safely past the vane, she will fall into the failure bin. The vane, which is driven by a randomly pulsed DC motor is the first impediment in this escape from Stanford.

2nd Feat: Binary Arithmetic

Upon successful completion of the first stage, the princess will arrive in a hopper attached to a robot arm. The hopper can either dump the princess into the failure pit, or transfer her safely into a tray attached to a catapult (see catapult, Figure: overview). What ultimately determines the princess’ fate in this stage is whether the user successfully solves a problem involving binary arithmetic. The way in which the arithmetic game is presented is by having 2 rows with 4 LEDs representing the two binary numbers. A third row represents the result (see Stage 2 Binary Addition, Figure: overview).

The two binary numbers in the first and second row are randomly assigned by our C32’s program. Whenever the user presses one of four buttons in front of the game set, the corresponding LED will light up. The user may make changes to they’re answer up until they press the enter button. If the user presses the ‘Enter’ button after calculating the answer, the result will be transferred to C32 to compare it with the correct answer. Forming the binary numbers are 12 LEDs composed of 3 rows, driven by 5833 serial to parallel driver.

The 5833 is a 32-bit, latched, serial to parallel converter, capable of directly driving LEDs from its parallel outputs. The reason we chose the 5833 was that it simplified our design by eliminating the need for two ULN2003 driver chips (for the LEDs) as well as two smaller 8-bit serial to parallel shift registers, each of which would have required its own clock and data input.
The 5833 is capable of being interfaced with four I/O lines, which is the same number that the two independent shift registers would need, however the 5833 has the advantage because in addition to possessing the capability of directly driving LEDs, the 5833 can interface up to 32 different devices, be they LED’s or other driver chips. This allows for future expansion of our game by allowing for more binary numbers or status LEDs without the need to add any more IC components.

If the user succeeds in figuring out the question, our robot arm having two DC motors will move the princess to a dish of the catapult. Otherwise, the hopper will dump her directly into the failure bin.

**3rd Feat: Catapult**

The user must be successful in catapulting the “princess” out of the device. The user must gauge the proper amount of force necessary to eject the “princess” from the device. If they fail, the “princess” will fall into the failure bin, otherwise if successful, the princess will escape the Stanford contraption.
Figure: Overview
Figure: Front View
GEMS OF WISDOM

1. **The Whole is Greater than the Sum of Its Parts:**

   This is a most valuable lesson that we learned from this project. We were more concerned with implementing each part rather than considering the integration of them at the software level and hardware level. In fact, we underestimated the amount of time that was necessary in order to integrate the individual parts into a functioning, cohesive whole. Thus, never underestimate the difficulty and time required to integrate modules into a collective whole.

   Also, we learned an interesting fact that even a trivial mistake at the circuit level can cause many great disasters. For instance, it was observed that an accidental disconnection/breakage of one of the wires connecting a momentary push button affected the proper operation of much of the rest of our breadboarded circuits. Thus, localized problems can have system-wide* repercussions of a circuit, from all out corruption to undesirable operation.

   *In our case, this single disconnection affected LEDs, motors, and even our comparator circuit.

2. **Modular Design:**

   The one of major reasons why we had a lot trouble in completely integrating the whole project was the difficulty in bridging between the creation of individual modules and an integrated whole. Each individual module was itself functional, however upon integration with another properly functioning module, would cease to perform their intended collective function. This lead to numerous hours attempting to debug the integration/compatibility errors, which was unfortunately not budgeted sufficient time.

3. **Orderliness:**

   Keeping circuits well planned out (opposed to rat’s nest) lets us to save time and makes it easy to debug the circuits.

4. **Finite testing:**

   All too often we would test a particular module too much, leading to component failure, as evidenced by the robotic arm’s hopper breaking a gear. Thus, it is better to test things to a mild degree and move on to other important things.

5. **Redundancy/Backups:**

   If possible have spare parts for things that might break in the course of testing.

6. **Simplicity:**

   As time grew short we concentrated (mentally & physically) on too many different ideas/tasks at once. It would have better served us if we stuck with certain problems and resolved them instead of worrying about how one was going to manage all the problems at once.
<table>
<thead>
<tr>
<th>Item desc.</th>
<th>Price($)</th>
<th>No.</th>
<th>Sub Total Price($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Photo-Transistor</td>
<td>0.50</td>
<td>3</td>
<td>1.50</td>
</tr>
<tr>
<td>IR-LED</td>
<td>0.35</td>
<td>3</td>
<td>1.05</td>
</tr>
<tr>
<td>RED LED</td>
<td></td>
<td>12</td>
<td>0.00*</td>
</tr>
<tr>
<td>Wire</td>
<td>1 $ per 10 ft</td>
<td>20 ft</td>
<td>2.00</td>
</tr>
<tr>
<td>Wire Connector1 female</td>
<td>0.70</td>
<td>6</td>
<td>4.20</td>
</tr>
<tr>
<td>Wire Connector2 male</td>
<td>0.15</td>
<td>6</td>
<td>0.90</td>
</tr>
<tr>
<td>Crimp pins</td>
<td>0.10</td>
<td>~40</td>
<td>4.00</td>
</tr>
<tr>
<td>IRLZ34N Transistor</td>
<td>0.80</td>
<td>3</td>
<td>2.40</td>
</tr>
<tr>
<td>LM324 Op-amp</td>
<td>0.25</td>
<td>1</td>
<td>0.25</td>
</tr>
<tr>
<td>LN339 Comparator</td>
<td>0.25</td>
<td>1</td>
<td>0.25</td>
</tr>
<tr>
<td>UCN5833 32 bit Serial to Parallel Latched Driver</td>
<td></td>
<td>1</td>
<td>0.00*</td>
</tr>
<tr>
<td>DC motor</td>
<td></td>
<td>3</td>
<td>0.00*</td>
</tr>
<tr>
<td>Foam Board 30*40 inch</td>
<td>4.34</td>
<td>4</td>
<td>17.36</td>
</tr>
<tr>
<td>Glue Gun Stick</td>
<td>6.46</td>
<td>1 pack</td>
<td>6.46</td>
</tr>
<tr>
<td>Transparent Plastic plate 1</td>
<td></td>
<td>1</td>
<td>8.49 + Tax</td>
</tr>
<tr>
<td>Transparent Plastic plate 2</td>
<td></td>
<td>1</td>
<td>13.49 + Tax (1.81 for both)</td>
</tr>
<tr>
<td>Various Resistors</td>
<td>~30</td>
<td></td>
<td>0.00*</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td>55.67</td>
</tr>
</tbody>
</table>

*obtained from the cabinet of freedom*
SCHEMATICS

1. Input part, Output parts (Success and Failure)

IR-LEDs and Photo-transistors for Input, Success Exit and Failure Exit
2. Feat 2 Circuits (Stage 2)

LED Drivers for Stage 2 (Binary Addition Game)
3. Feat 3 Circuit(Stage3): No circuit was specifically necessary.
4. Motor Drivers for Input, two Output exits, and the branch to Stage 2

DC MOTOR Drivers for the Vane of Stage 1, Grapper and Robot Arm

Note: Gate Motor Driver, not listed, but identical to the circuits seen below.
Stage One Pseudo Code:

Poll Infrared gate sensor to see if token has been inserted into slot.

- When token inserted
  - a) Turn on green LED. (Define as bit 0 of index 0 of array_indices_for_display*)
  - b) Turn on timer for global clock (this is the 60 second timer)
  - c) Turn on the rotator process, (the propeller that turns for a unknown amount of time, (from the user stand point, but is actually randomly actuated).
  - d) Poll the gate button to see if the gate needs to be raised.
    i. If the gate button is pushed take a time sample of this instance so we know when to time out of stage, aka T1_timeout.
    ii. If (T2_timeout – T1_timeout) > Timeout and failure sensor not tripped, then proceed to stage 2.
       Else, failed. Expire the rotator process, either way once we’ve completed/failed stage 1.
  - e) Poll the failure sensor (this should actually be broken out of this routine to make it poll for failure regardless of what stage we are in).

*Note: array_indices_for_display is composed of 5 nibbles, nibbles 0 and 1 are reserved for general status LEDs such as the LED indicating the game has started, as well as whether your mission has failed, or if you’ve won, for now we have a surplus of status bits, but allows us to add more LED functionality, timer permitting. Nibbles 2,3,4 are reserved for the binary math portion of the game, where Nibble 2 is the upper number, Nibble 3 is the lower number and nibble 4 is the result of Nibble 2 + Nibble 3.
Stage Two Pseudo Code:

Requisite tasks in this stage are opening the gripper and the swinging arm. There are two modes of operation for the robotic arm, one is if the user correctly computes the binary result and the second is if they compute the result incorrectly.

Success in binary
- Swing out arm, then fire gripper after delay of time X.

Failure in binary
- Fire gripper for amount of time Y.

Provided failure bin detection is not activated from pendulums, from them striking the token, we need to wait for time out. Say time out of feat one is 8 seconds (8000 ticks of a 1ms timer). Upon time out, we need to wait for binary math result to occur. Once enter has been pressed, we need to store that time as T1 and fire off the swing arm. As the code loops we need to compute $\Delta T = T2' - T1$ until $T2$, s.t. $\Delta T =$ the firing time for the gripper.

We also need to send out two random 4-bit number to the LEDs:
So we need to generate a number between 0 and 15 $\rightarrow$ rand()%16 yields the appropriate range.

Get upper $\leftarrow$ rand()%16
Get lower $\leftarrow$ rand()%16

Place these into the array_indices_for_display by calling the appropriate helper function*. 

Compute upper + lower = result&0x0F to ensure it stays 4-bit, (we don’t care about any carries into the next nibble).

Now compare this result when the user has pressed the enter key. And either direct the dumper arm to transfer the token into the catapult’s dish, otherwise if they incorrectly computed the result, dump the token into the failure bin.

In order to see what buttons were pressed take a debounced sample of port A/D and EOR the result with the previous button history to get the updated button states. We need to do this b/c the buttons are momentary switches so we need to sample the buttons for transitions of 0 $\rightarrow$ 1, b/c this is when a button is released.

*Note: creates a 20-bit array
int index=0
int bit
int nibble=0
for( ; nibble < 4; nibble++)
{
 for(bit=0 ; bit < 4; bit++)
 {
     bitarray[index]=binary_numbers[array_indices[nibble]][bit];
     index++;
 }
We’ll need to write some logic to time the gripper and arm so that the gripper dumps only when the arm has reached its limit and then fire off the gripper. For instance, fire of arm motor and let it run for 8 seconds, however after 4 seconds, fire off the gripper arm, b/c the arm would have reached its limit by 4 seconds, then let both motor processes expire on their own (so that the arm returns to it resting position).
Stage Three Pseudo Code:

The only thing we need to look at here is to poll the success bin’s sensor until an event happens. We won’t need to poll the failure bin’s sensor since we will have out sourced this to a separate function nor will have to check the global time, as this will also be in a separate function.

However, upon success detection, the robot arm can make some swinging out maneuvers like its “waving” at you.

A note on failure and global time expiration. If at any time the failure sensor is tripped, all other feats will be locked out so you can no longer “try” to play the game until the token is reinserted into the gate, at which time any state variables which were modified up to the point of failure detection will be reset to their default values. The same goes for a global timeout (aka “60 seconds is up”), when time is up and the player has not won the game, the game will lockout all stages from processing until the token is reinserted into the slot.
Motor Driving:
In order to drive any requisite motors we’ll be formulating a struct data structure of motor properties, these properties will indicate such things as whether the motor is on, the bit on the port it exists on, whether we need to start up a motor, etc. There are going to be approximately five motors, so we’ll store this struct in an array so we can easily cycle through all the motors and tend to each one’s needs in an appropriate manner. We’ll also make a distinction between non periodic motors and periodic motors. Periodic motors are used for things such as the driving the vane, while non periodic motors are ones which are called only to perform a particular function only for a certain amount of time.

We’ll also have to worry about coordination of motors, but this will be handled by a “higher level” function.

typedef struct
{
  unsigned char initialize; //whether or not to start up motor
  unsigned char motorID; //Which motor we are talking about
  unsigned char timerChannel; //Which channel we are using
  unsigned int onTime;  //In milliseconds
  signed char state;  //On or Off
  unsigned char taskPeriodic;
  unsigned int period;
  
  unsigned char enabled; //for periodic motors
}
} Motor;

The initializer will activate the timer for the motor process.

Question: does an uninitialized timer return expired? No, it returns NOT_EXPIRED.

0-false, “not expired”
1-true, “has expired”

if(state == on)
  check if expired
    if expired – set state to off
    else
      - turn on motor channel
else //state == off
  check if expired
    if expired – do nothing
    else
      - turn off motor
LED Driving with 5833, 32-bit Serial-Input Latched Driver:

![Timing Diagram and Condition of 5288](image)

The pseudo code is based on this timing diagram and condition. We wrote three categorized functions:

1) Making a data pack for LED diving: `assembleSerialData()`
2) Sending data: `sendBit()`, `sendSerializedData()`
3) Interfacing function: `ClockPulse()`, `DataInPulse()`, `StrobePulse()`, `OutputPulse()`

Those interface functions are for maintaining minimum data high time or low time. (high = 5V, low = 0V)

```cpp
void assembleSerialData (int *array_indices)
{
    // This function takes an array to store 20-bit binary numbers.
    // The 20-bit number is to represent 5 nibbles for LED driving.
    // An array, serialized[20] is defined as global.

    Repeat
        Put the data into serialized[] array
    Until the serialized array is full
}

void sendSerialData(unsigned char *Bits)
{
    Sending the array, Bit[20] by using the function `sendBits()`
}```
void sendBits(unsigned char bit)
{
    // This function is implemented based on the timing diagram as above-mentioned.
    // The following constant is predefined.
    // T0=0, T1=100, T2=200, T3=300, T4=400, T5=500, T6=600;

    Local variable: int time
    Repeat For TOTAL_TIME
        If time == T0, Put DataIn from the C32 output to 5833;
        Else if time == T1, Clock high
        Else if time == T2, DataIn low
        Else if time == T3, Clock low and then Strobe high;
        Else if time == T4, Strobe low
        Else if time == T5, OutputEnable low
        Else if time == T6, OutputEnable high
    }

    // interfaacing functions

void ClockPulse(unsigned int clockState)
{
    If clockState == High, clock high (PORTT BIT0)
    Else clock low
}

void DataPulse(unsigned int dataInState)
{
    If dataInState == High, DataIn high (PORTT BIT1)
    Else DataIn low
}

void StrobePulse(unsigned int strobeState)
{
    If strobeState == High, strobe high (PORTT BIT2)
    Else strobe low
}

void OutputEnable (unsigned int enableState)
{
    If enableState == High, enableState high (PORTT BIT3)
    Else enableState low
}
SOURCE CODE

/* %%%%%%%%%%%%%%%% Initializers %%%%%%%%%%%%%%%%%% */
/* %%%%%%%%%%%%%%%% Prototypes/Header file %%%%%%%%%%%%% */
/*
Initialize the properties of all the motors.
*/
static void initMotorPropertiesArray(void)
/* Setup the timer to have a rate
   Probably not a good idea to set the rate of
   the timers to something else during operation,
   only at initialization.
*/
void InitializeTimerRate(void)

/* Sets up specified timer channel with the tick count expiration*/
void setupTimerChannel(unsigned char channel, unsigned int ticks)

/* Sets up all bits on Port as output
   Sets all output bits LOW, so the motors don't turn on!!!
*/
void SetupAsOutputPortM(void)

/* Makes port an output port
   for the 5833
*/
void SetupAsOutputPortT(void)

/* Sets up all bits on Port as input */
void SetUpPortADC(void)

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
/* %%%%%%%%%%%%%%%% Motor Functions %%%%%%%%%%%%%%%%%% */
/* %%%%%%%%%%%%%%% Prototypes/Header file %%%%%%%%%%%%%%% */
/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */

/* Specifies whether motor is on or off
   We are not using PWM here because the motor
   is simply on or off, we want to have max speed.
*/
void MotorScheduler(void)

/* Specify which motor to turn on or off */
void SendMotorCommand (unsigned char motorID, unsigned state)

/* Handles scheduling tasks for non-periodic running of motors */
void nonPeriodicMotor (int motor)

/* Handles scheduling tasks for the periodic running of motor(s)
   Note: PWM won't cut it here b/c we may want to have irregular (or random) periods,
   as well as periods that extend into the seconds.
*/
void periodicMotor (int motor)

/*
This routine is responsible for directing the robotic arm timing sequence
Note: This routine had to be modified when it was found that our robot arm's
gripper portion gear mechanism failed and the spring on the arm weakened.
*/
void taskScheduler(void)
void assembleSerialData(int *array_indices)

void sendSerialData(unsigned char *Bits)

void sendBit(unsigned char bit)

void ClockPulse (unsigned int clockState)

void DataInPulse (unsigned int DataInState)

void StrobePulse (unsigned int StrobeState)

void OutputEnable (unsigned int enableState)

void readButtons(void)

/* Responsible for:
   - generating the binary numbers (once only)
   - sending out numbers to the LED display
   - comparing result of any user input
   - echoing any key presses back the LED display, update LEDs only when there's a change in state of one of the key presses. */
void binaryGenerator(void)
/* responsible for running the first stage. */
void passThruRotator(void)
/* final stage, senses win condition, or failure. */
void catapult(void)

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
/* %%%%%%%%%%%%%%%%%%%%%%%%% Helper Functions %%%%%%%%%%%%%%%%%%%%%%%%% */
/* %%%%%%%%%%%%%%%%% Prototypes/Header file %%%%%%%%%%%%%%%%%%%%%%%%% */
/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
/* Check to see if the timer channel has expired */
unsigned char checkIsExpired(unsigned char channel)
/* returns the value of port T */
unsigned char sampleOfPortT(void)
/* returns the value of port AD */
unsigned char sampleOfPortAD(void)
/* send back the port's debounced value */
unsigned char debouncedSampleOfPortAD(void)
/* updates LEDS with the new stuff
if nothing to update, it wont refresh the LEDs. */
void refreshLEDs(void)
/* Set up robot arm to either chuck into pit, or place into catapult's dish
  if won==1, they won
  if won==0, the lost */
void sendMotor(unsigned char won)
/* clears out the binary math LEDS */
void cleanUp(void)
Some notes on additional I/O ports
Port E is DDRE and PTE
Port A/D is PTAD (output) and PTIAD (input)
*/

#include <stdio.h>
#include <ME218_C32.h>
#include <timers12.h>
#include "ADS12.h"
#include <stdlib.h>
#include <string.h>
#include "prototypes.h"

// Time Constants for providing 8533 w/ the correct sequence of events.
#define T0  0
#define T1  100
#define T2  200
#define T3  300
#define T4  400
#define T5  500
#define T6  600

// Number of display LEDs
#define NUM_LED     12

#define PORT_HI                 1
#define PORT_LO                -1
#define PORT_NO_CHANGE          0

// Used in 8533 to denote logic levels
#define HIGH        1
#define LOW         0

// Motors
#define MOTOR_OFF               0
#define MOTOR_ON                1
#define MOTOR_DISABLED          0
#define MOTOR_ENABLED           1
// remember to Update this, as you add motors!!!!
#define NUMBER_OF_MOTORS        4
#define GATE_MOTOR              0
#define GRIPPER_MOTOR           1
#define ARM_MOTOR               2
#define ROTATOR_MOTOR           3
#define MOTOR_4                 4
#define MOTOR_5                 5

// Timers
#define TIMER_CHANNEL_0         0  //reserved for GATE_MOTOR
#define TIMER_CHANNEL_1         1  //reserved for GRIPPER_MOTOR
#define TIMER_CHANNEL_2         2  //reserved for ARM_MOTOR
#define TIMER_CHANNEL_3         3  //reserved for Rotator
#define TIMER_CHANNEL_4         4
#define TIMER_CHANNEL_5         5
#define TIMER_CHANNEL_6         6
#define TIMER_CHANNEL_7         7  //reserved for debounce
#define DEBOUNCE_TIME           30  //30 ms
#define NO_TIME 0
#define GATE_TIME 2000 //empirically determined
#define GRIPPER_TIME 2500 //empirically determined
#define ARM_TIME 7000 //empirically determined
#define DONT_INITIALIZE_MOTOR 0
#define START_MOTOR_PROCESS 1
#define TASK_IS_PERIODIC 1
#define TASK_IS_NOT_PERIODIC 0
#define PERIOD_0 0
#define PERIOD_1MS 1
#define TIMER_EXPIRATION_TIME 8

/* Used by the Motor Scheduler to write the correct bit pattern to port M.
static unsigned char PortHi[] = {BIT0HI, BIT1HI, BIT2HI, BIT3HI, BIT4HI, BIT5HI, BIT6HI};
static unsigned char PortLo[] = {BIT0LO, BIT1LO, BIT2LO, BIT3LO, BIT4LO, BIT5LO, BIT6LO};
*/

These bit patterns are used to convert the decimal numbers to their binary equivalents when sending out data to the 5833 serial to parallel converter

unsigned char binary_numbers[4][4] =
{ {0, 0, 0, 0}, {0, 0, 0, 1}, {0, 0, 1, 0}, {0, 0, 1, 1},
{0, 1, 0, 0}, {0, 1, 0, 1}, {0, 1, 1, 0}, {0, 1, 1, 1},
{1, 0, 0, 0}, {1, 0, 0, 1}, {1, 0, 1, 0}, {1, 0, 1, 1},
{1, 1, 0, 0}, {1, 1, 0, 1}, {1, 1, 1, 0}, {1, 1, 1, 1} );

static unsigned char serialized[20]; //GLOBALIZED
static unsigned char was_button_pressed[6]; //GLOBALIZED
static unsigned char compositeHexButton; // Which buttons were pressed
static int array_indices_for_display[5] = {0, 0, 0, 0, 0}; //GLOBALized
static unsigned char globalAlert_failed=0;
static unsigned char won_stage1=0;
static unsigned char won_stage2=0;
static unsigned int start_time=0;
static unsigned int stage1_lockout=0;
static unsigned int stage2_lockout=0;
static unsigned int stage3_lockout=0;
static unsigned char lockout1_stage2=0;
static unsigned int lockedout1_stage1=0;
static unsigned int lockedout2_stage1=0;
static unsigned int lockedout3_stage1=0;
static unsigned int binaryMathComplete=0;
static unsigned char updateLEDs=0;

typedef struct
{
    unsigned char initialize; //whether or not to start up motor
    unsigned char motorID; //Which motor we are talking about
    unsigned char timerChannel; //Which channel we are using
    unsigned int onTime; //In milliseconds
    signed char state; //On or Off
    unsigned char taskPeriodic;
    unsigned int period;
    unsigned char enabled; //for periodic motors
}
//unsigned int lowTime;
static Motor motor_list[NUMBER_OF_MOTORS];

#include <math.h>

/*
 ******************* Main Loop *******************
 */

/*Main Loop: runs all the routines in a non blocking manner */
void main (void)
{
    //variables for the artificial delay
    unsigned int i, j;

    // Initialize the ports, timers, and motor properties.
    SetUpPortADC();
    SetupAsOutputPortM();
    SetupAsOutputPortT();
    InitializeTimerRate();
    initMotorPropertiesArray();

    //pause before actually starting any execution so we have time
    //to turn on power supply etc.
    for (i=0; i<10000; i++)
    {
        for(j=0; j<1000; j++);
        {
        }
    }

    while(1)
    {

        //testGateButton();
        //stage one:
        if (lockedout1_stage1==0)
        {
            passThruRotator();
        }

        //stage two:
        if(binaryMathComplete==0)
        {
            binaryGenerator();
        }

        //stage two, b
        if( binaryMathComplete == 1)
        {
            taskScheduler();
        }

        //testTimer();

        //stage three
        //catapult();

        //testGateButton();
        //passThruRotator();
        //taskScheduler();
//testTimer();
//binaryGenerator();
//refreshLEDs();
//testPrintOutRandomNumbers();

//general tasks
MotorScheduler();
refreshLEDs();

} //InitializeLED(); // clear LEDs for Stage2
} // main

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
/* %%%%%%%%%%%%%%%% Initializers %%%%%%%%%%%%%%%%%%
/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */

/* Initialize the properties of all the motors. */
static void initMotorPropertiesArray(void)
{
    Motor gate = {DONT_INITIALIZE_MOTOR,GATE_MOTOR, TIMER_CHANNEL_0, GATE_TIME, MOTOR_OFF, TASK_IS_NOT_PERIODIC, PERIOD_0, MOTOR_DISABLED};
    Motor gripper = {DONT_INITIALIZE_MOTOR, GRIPPER_MOTOR, TIMER_CHANNEL_1, GRIPPER_TIME, MOTOR_OFF, TASK_IS_NOT_PERIODIC, PERIOD_0, MOTOR_DISABLED};
    Motor arm = {DONT_INITIALIZE_MOTOR, ARM_MOTOR, TIMER_CHANNEL_2, ARM_TIME, MOTOR_OFF, TASK_IS_NOT_PERIODIC, PERIOD_1MS, MOTOR_DISABLED};
    Motor rotator = {DONT_INITIALIZE_MOTOR, ROTATOR_MOTOR, TIMER_CHANNEL_3, 2000, MOTOR_OFF, TASK_IS_PERIODIC, 2500, MOTOR_DISABLED};
    motor_list[0] = gate;
    motor_list[1] = gripper;
    motor_list[2] = arm;
    motor_list[3] = rotator;
}

/* Setup the timer to have a rate
   Probably not a good idea to set the rate of
   the timers to something else during operation,
   only at initialization. */
void InitializeTimerRate(void)
{
    TMRS12_Init(TMRS12_RATE_1MS);
}

/* Sets up specified timer channel with the tick count expiration*/
void setupTimerChannel(unsigned char channel, unsigned int ticks)
{
    TMRS12_InitTimer(channel, ticks);
    //while(TMRS12_EXPIRED!=TMRS12_IsTimerExpired(TIMER_CHANNEL));
    //return;
}

/* Sets up all bits on Port as output
   Sets all output bits LOW, so the motors don't turn on!!! */
void SetupAsOutputPortM(void)
{
    DDRM = 0xFF;
    PTM = 0x00;
}

/*
Makes port an output port
for the 5833
*/
void SetupAsOutputPortT(void)
{
    DDRT = 0xFF;
    PTT = 0x00;
}

/* Sets up all bits on Port as input */
void SetUpPortADC(void)
{
    ADS12_Init("IIIIIIII");
}

// Initializing LEDs for stage2
// Clear all LEDs
void InitializeLED(void)
{
    int i;
    ClockPulse(LOW);
    DataInPulse(LOW);
    StrobePulse(LOW);
    OutputEnable(HIGH);
    for(i=0;i<NUM_LED;i++)
    { // Clear all LEDs
        ClearLED();
        return;
    }
}

/*****************************/
/**** Motor Functions ***********/
/*****************************/

/* Specifies whether motor is on or off
We are not using PWM here because the motor
is simply on or off, we want to have max speed.
*/
void MotorScheduler(void)
{
    //static int active_motors = 8;
    int motor;
    //Motor currentMotor;
    for(motor=0; motor < NUMBER_OF_MOTORS; motor++)
    {
        if (motor_list[motor].taskPeriodic == TASK_IS_NOT_PERIODIC)
        {
            //printf("non-periodic\r");
            nonPeriodicMotor(motor);
        }
        else
        {
            //printf("periodic\r");
            periodicMotor(motor);
        }
    }
}
/* Handles scheduling tasks for non-periodic running of motors */
void nonPeriodicMotor (int motor)
{
    //Initialize motor:
    if (motor_list[motor].initialize==START_MOTOR_PROCESS)
    {
        motor_list[motor].state=MOTOR_ON;
        setupTimerChannel (motor_list[motor].timerChannel, motor_list[motor].onTime);
        motor_list[motor].initialize=DONT_INITIALIZE_MOTOR;
    }
    else
    {
        // for simple, non-periodic motor control
        if (motor_list[motor].state==MOTOR_ON)
        {
            if(checkIsExpired(motor_list[motor].timerChannel)!=1)
            {
                //set Port M bit motor_list[motor].motorID active
                SendMotorCommand (motor_list[motor].motorID, motor_list[motor].state);
            }
            else // Time has expired
            {
                motor_list[motor].state=MOTOR_OFF;
            }
        }
        else // Motor is in off state
        {
            //set Port M bit motor_list[motor].motorID inactive
            SendMotorCommand (motor_list[motor].motorID, motor_list[motor].state);
        }
    }
}

/* Handles scheduling tasks for the periodic running of motor(s)
Note: PWM won't cut it here b/c we may want to have irregular (or random) periods,
as well as periods that extend into the seconds.
*/
void periodicMotor (int motor)
{
    //Initialize motor:
    if (motor_list[motor].enabled==MOTOR_ENABLED)
    {
        if (motor_list[motor].initialize==START_MOTOR_PROCESS)
        {
            motor_list[motor].state=MOTOR_ON;
            //printf("Initialization...
");
            setupTimerChannel (motor_list[motor].timerChannel, motor_list[motor].period);
            motor_list[motor].initialize=DONT_INITIALIZE_MOTOR;
        }
        else
        {
            // for simple, non-periodic motor control
            if (motor_list[motor].state==MOTOR_ON)
            {
                //printf("Motor ON
");
                if(checkIsExpired(motor_list[motor].timerChannel)!=1)
                {
                    //set Port M bit motor_list[motor].motorID active
                }
            }
        }
    }
}
SendMotorCommand (motor_list[motor].motorID, motor_list[motor].state);
} else // Time has expired
{
    motor_list[motor].state=MOTOR_OFF;
    motor_list[motor].initialize=START_MOTOR_PROCESS;
}
}
else // Motor is in off state
{
    //printf("Motor OFF\n");
    if(checkIsExpired(motor_list[motor].timerChannel)!=1)
    {
        //set Port M bit motor_list[motor].motorID active
        SendMotorCommand (motor_list[motor].motorID, motor_list[motor].state);
    } else // Time has expired
    {
        motor_list[motor].state=MOTOR_ON;
        motor_list[motor].initialize=START_MOTOR_PROCESS;
    }
}
}
else
{
    motor_list[motor].state=MOTOR_OFF;
    SendMotorCommand (motor_list[motor].motorID, motor_list[motor].state);
}
}

/* Specify which motor to turn on or off */
void SendMotorCommand (unsigned char motorID, unsigned state)
{
    if (state==MOTOR_ON)
    {
        PTM = PortHi[motorID] | PTM;
    }
    else
    {
        PTM = PortLo[motorID] & PTM;
    }
}

/*
This routine is responsible for directing the robotic arm timing sequence
Note: This routine had to be modified when it was found that our robot arm's
gripper portion gear mechanism failed and the spring on the arm weakened.
*/
void taskScheduler(void)
{
    static unsigned char lockout1=0;
    static unsigned char lockout2=0;
    static unsigned char lockout3=0;
    static unsigned int arm_start_time;
    //static unsigned int arm_end_time;
    static unsigned int swivel_time=9000; //make it double the required amount
    static unsigned int gripper_start_time=3000; //gripper drops midway between
    static unsigned int current_time;

    if ((binaryMathComplete==1) && (lockout1==0))
{  //printf("Entering...");  //if (motor_list[2].taskPeriodic==TASK_IS_NOT_PERIODIC)  //if (motor_list[2].taskPeriodic==TASK_IS_PERIODIC)  if (won_stage2==1)  {  if(lockout2==0)  {  printf("Get arm start time...");  arm_start_time=TMRS12_GetTime();  lockout2=1;  motor_list[ARM_MOTOR].initialize=START_MOTOR_PROCESS;  }  //motor_list[ARM_MOTOR].enabled=MOTOR_ENABLED;  current_time=TMRS12_GetTime();  if (((current_time-arm_start_time) >= gripper_start_time)&&(lockout3 == 0))  {  printf("Starting gripper motor...");  motor_list[GRIPPER_MOTOR].initialize=START_MOTOR_PROCESS;  lockout3=1;  }  /*if ((current_time-arm_start_time) >= swivel_time)  {  printf("Stopping arm motor...");  motor_list[ARM_MOTOR].enabled=MOTOR_DISABLED;  lockout1=1;  }*/  if ((current_time-arm_start_time) >= swivel_time)  {  //motor_list[ARM_MOTOR].enabled=MOTOR_DISABLED;  lockout1=1;  }  else  {  printf("Failed in binary");  motor_list[GRIPPER_MOTOR].initialize=START_MOTOR_PROCESS;  lockout1=1;  }  }  }  /* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */  /* %%%%%%%%%%%%%%%% Serial to Parallel        %%%%%%%%%%% */  /* %%%%%%%%%%%%%%%% Data Output Functionality %%%%%%%%%%% */  /* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */  /* * %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% **/  /* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% **/  /* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% **/  /* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% **/  /* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% **/  /* Creates an array of 20 bits from the specified binary numbers. */  void assembleSerialData(int *array_indices)  {  int bit;  int nibble;  int index=0;  //unsigned char serialized[12]; ////////////*GLOBALIZED  for (nibble=0; nibble < 5; nibble++)  {  for (bit=0; bit < 4; bit++)  {  
}
serialized[index]=binary_numbers[array_indices[nibble]][bit];
    index++;
}
//return serialized;
}

/* Sends a sequence of 20 bits out to the 5833 */
void sendSerialData(unsigned char *Bits)
{
    static unsigned int length=20;
    unsigned int bit;
    for (bit=0; bit < length; bit++)
    {
        sendBit(Bits[bit]);
    }
}

/* Accepts a value which is either zero or one. 
Send one bit to the 5833 chip's serial input line. 
This function conforms to the timing diagram specified in the 5833's data sheet. */
void sendBit(unsigned char bit)
{
    unsigned int time;
    unsigned int TOTAL_TIME=650;
    //static unsigned int T0=0, T1=100, T2=200, T3=300, T4=400, T5=500, T6=600;
    for(time=0; time < TOTAL_TIME; time++)
    {
        switch (time)
        {
            case T0: DataInPulse(bit);
            break;
            case T1: ClockPulse (HIGH);
            break;
            case T2: DataInPulse(LOW);
            break;
            case T3: ClockPulse (LOW);
                StrobePulse(HIGH);
            break;
            case T4: StrobePulse(LOW);
            break;
            case T5: OutputEnable(LOW);
            break;
            case T6: OutputEnable(HIGH);
            break;
        } //end of switch
    } //end of for
}

/*
5833 Interfacing: Toggles the clock line depending upon the input parameter. */
void ClockPulse (unsigned int clockState)
{
    if (clockState == HIGH)
        PTT = PTT | BIT0HI; // make clock pin high
    else
        PTT = PTT & BIT0LO; // make clock pin low
}
void DataInPulse (unsigned int DataInState)
{
    if (DataInState == HIGH)
        PTT = PTT | BIT1HI; //
    else
        PTT = PTT & BIT1LO; //
}

void StrobePulse (unsigned int StrobeState)
{
    if (StrobeState == HIGH)
        PTT = PTT | BIT2HI;
    else
        PTT = PTT & BIT2LO;
}

void OutputEnable (unsigned int enableState)
{
    if (enableState == HIGH)
    {
        PTT = PTT | BIT3HI; // Enable High
    }
    else
    {
        PTT = PTT & BIT3LO; // Enable Low
    }
}

/* Returns which buttons were pressed. */
void readButtons(void)
{
    static unsigned char num_of_buttons=6;
        // bit0  bit 1  bit 2  bit 3  enter  gate
    buttons
    static unsigned char button_mask[] = {BIT0HI, BIT1HI, BIT2HI, BIT3HI, BIT4HI, BIT5HI};
    static unsigned char last_button_state[] = {BIT0HI, BIT1HI, BIT2HI, BIT3HI, BIT4HI, BIT5HI};
    //unsigned char was_button_pressed[6]; ////////////*GLOBALIZED

    unsigned char port;
    unsigned int button;

    compositeHexButton = 0x00;
    port = debouncedSampleOfPortAD();

    for(button=0; button < num_of_buttons; button++)
state was low
if
{{(port&button_mask[button])==button_mask[button]}&&(last_button_state[button]==0x00)}
{
    //then button was pressed & released
    was_button_pressed[button]=HIGH;
    compositeHexButton = compositeHexButton | port&button_mask[button];
}
else
{
    was_button_pressed[button]=LOW;
}
last_button_state[button]=port&button_mask[button];
} //return was_button_pressed;

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
/* %%%%%%% The Three Feats/Stages Implementation %%%%%%%% */
/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
/* responsible for running the first stage. */
void passThruRotator(void)
{
    unsigned char portAD;
    unsigned char failureSensor;
    unsigned char gateButton;
    unsigned char gateSensor;
    //static unsigned int lockedout1_stage1=0;
    //static unsigned int lockedout2_stage1=0;
    //static unsigned int lockedout3_stage1=0;
    static unsigned int T1_timeout;
    static unsigned int stage1_timeout=8000;
    unsigned int random;
    if(lockedout1_stage1 == 0)
    {
        printf("entering first stage...");
        //srand(TMRS12_GetTime());
        //random=(rand()%16)*100;
        //printf(" random: %d", random);
        //motor_list[ROTATOR_MOTOR].onTime=random;
        gateButton = debouncedSampleOfPortAD() & BIT4HI; // masking port AD4
        portAD = sampleOfPortAD();
        gateSensor = portAD & BIT7HI;
        failureSensor = portAD & BIT6HI;
        if ((gateSensor==0)&&(lockedout3_stage1==0))
        {
            printf("object inserted");
            motor_list[ROTATOR_MOTOR].enabled=MOTOR_ENABLED;
            T1_timeout=TMRS12_GetTime();
            lockedout3_stage1=1;
        }
        if ((gateButton == 0)&&(gateSensor == 0)) // press the button but lockout if
nothing in slot
        {
            motor_list[GATE_MOTOR].initialize = START_MOTOR_PROCESS; // drive the motor
//set go LED to ON
array_indices_for_display[1] = 0x01;
updateLEDs=1;
}

// Start up the 1 minute timer
if ((gateSensor == 0)&&(lockedout2_stage1==0))
{
    start_time=TMRS12_GetTime();
    lockedout2_stage1=1;
}
if (failureSensor == 0)
{
    motor_list[ROTATOR_MOTOR].enabled=MOTOR_DISABLED;
    globalAlert_failed=1;
    //lockedout1_stage1=1; uncomment
    array_indices_for_display[1] = 0x08; //indicate failure on LED display
    updateLEDs=1;
    //set failure LED
}
if (((TMRS12_GetTime() - T1_timeout) > stage1_timeout)&&(globalAlert_failed==0))
{
    motor_list[ROTATOR_MOTOR].enabled=MOTOR_DISABLED;
    //lockedout1_stage1=1; uncomment
    won_stage1=1;
    array_indices_for_display[1] = 0x00;
    updateLEDs=1;
    //set
}

/* Responsible for:
- generating the binary numbers (once only)
- sending out numbers to the LED display
- comparing result of any user input
- echoing any key presses back the LED display, update LEDs only when there's
  a change in state of one of the key presses.
*/
void binaryGenerator(void)
{
    static unsigned int upper;
    static unsigned int lower;
    //static int array_indices_for_display[3]; ///////////*Globalized
    //static unsigned char lockout1_stage2=0;
    static unsigned char user_entry[4]= {0, 0, 0, 0};
    static unsigned char button_history=0x00;
    static unsigned char result;
    //unsigned char was_button_pressed[6];
    static unsigned int time1, time2;
    //int i;
    int found=0;
    if (!lockout1_stage2)
    {
        //time1=TMRS12_GetTime(); //used for testing seed
        srand(TMRS12_GetTime()); //set a new random seed
        upper=rand()%16; //get the upper random 4 bit number
        lower=rand()%16; //get the lower random 4 bit number
        result = (upper+lower)&0x0F; //Add upper & lower, and truncate it to a 4 bit
        number
printf(" upper: %d", upper);
printf(" lower: %d", lower);
printf(" result: %d", result);
lockout1_stage2 = 1;
//array_indices_for_display[0] = status_LEDS; //
array_indices_for_display[2] = upper; // Upper number for display on LEDs
array_indices_for_display[3] = lower; // Lower number for display on LEDs
array_indices_for_display[4] = 0; // Clear out the result LEDs as the user needs
to compute these.
//array_indices_for_display[4] = result; //used for testing result
assembleSerialData(array_indices_for_display); // make an array of bits.
sendSerialData(serialized); // send out the array of bits to the 5833.
/*for (i=0; i < 20; i++)
{
    printf("%d", serialized[i]);
}
*/
/*time2=TMRS12_GetTime();
if((time2-time1)>10000)
{
    time1=TMRS12_GetTime();
    lockout=0;
}*/
// get User input...
readButtons();

button_history = button_history ^ compositeHexButton; // update buttons as need be,
if a button was pressed twice
//printf(" buttons pressed: %d", button_history);
// echo result back to LEDs
// cycle over only the bit buttons, excluding the enter and gate.
// for (i=0; i < 4; i++)
/*{
}
*/
if (compositeHexButton!=0x00)
{
    if ((compositeHexButton&BIT4HI) != 0x00) // pressed enter button
    {
        printf(" Enter Button Pressed ");
        // compare user's with computed result.
        printf(" User Entry %d", (button_history&0x0F));
        if (result == (button_history&0x0F))
        {
            // send out successful motor sequence
            sendMotor(1); // send win condition
            cleanUp();
            // Feat=3;
        }
        else
        {
            // send out failure motor sequence
            sendMotor(0); // send loss condition
            cleanUp();
            // Feat=FAILED;
        }
    }
    else
    {
        printf(" buttons pressed: %d", button_history);
        array_indices_for_display[4]=button_history&0x0F;
        assembleSerialData(array_indices_for_display);
    }
}
sendSerialData(serialized);
}
}

/* final stage, senses win condition, or failure. */
void catapult(void)
{
    unsigned char portAD;
    unsigned char successSensor;
    unsigned char failureSensor;

    portAD = sampleOfPortAD();
    successSensor = portAD & BIT5HI;
    failureSensor = portAD & BIT6HI;

    if (successSensor == 0)
    {
        //doDance();
        array_indices_for_display[1] = 0x04; //success LED
        updateLEDs=1;
    }
    if( failureSensor == 0)
    {
        globalAlert_failed=1;
        //lockout anything?
        array_indices_for_display[1] = 0x08; //indicate failure on LED display
        updateLEDs=1;
    }
}

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
/* %%%%%%%%%%%%%%%% Helper Functions %%%%%%%%%%%% */
/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */

/* Check to see if the timer channel has expired */
unsigned char checkIsExpired(unsigned char channel)
{
    return (TMRS12_EXPIRED==TMRS12_IsTimerExpired(channel));
}

/*returns the value of port T*/
unsigned char sampleOfPortT(void)
{
    unsigned char PortT;
    PortT=PTT;
    return PortT;
}

/*returns the value of port AD */
unsigned char sampleOfPortAD(void)
{
    unsigned char PortAD;
    PortAD = PTIAD;
    return PortAD;
}

/* send back the port's debounced value */
unsigned char debouncedSampleOfPortAD(void)
{
    static unsigned char startTimer = 1;
static unsigned char portValue;

if (startTimer==1)
{
    setupTimerChannel(TIMER_CHANNEL_7, DEBOUNCE_TIME);
    startTimer=0;
    portValue = sampleOfPortAD();
    return portValue;
}
else
{
    if(checkIsExpired(TIMER_CHANNEL_7))
    {
        setupTimerChannel(TIMER_CHANNEL_7, DEBOUNCE_TIME);
        portValue = sampleOfPortAD();
        return portValue;
    }
    else
    {
        return portValue;
    }
}

/*
Updates LEDs with the new stuff 
if nothing to update, it wont refresh the LEDs. */
void refreshLEDs(void)
{
    if(updateLEDs == 1)
    {
        assembleSerialData(array_indices_for_display);
        sendSerialData(serialized);
        updateLEDs=0;
    }
}

/* Set up robot arm to either chuck into pit, or place into catapult's dish 
    if won==1, they won 
    if won==0, the lost */
void sendMotor(unsigned char won)
{
    if (won==1)
    {
        //motor_list[2].enabled=MOTOR_ENABLED; 
        //motor_list[2].taskPeriodic=TASK_IS_PERIODIC; 
        //motor_list[ARM_MOTOR].onTime=5; 
        binaryMathComplete=1; 
        won_stage2=1;
    }
    else
    {
        //motor_list[2].enabled=MOTOR_ENABLED; 
        //motor_list[2].taskPeriodic=TASK_IS_NOT_PERIODIC; 
        binaryMathComplete=1; 
        won_stage2=0;
    }
}
/* clears out the binary math LEDS */
void cleanUp(void)
{
  // clear up binary quiz display
  array_indices_for_display[2] = 0x00; // Upper number
  array_indices_for_display[3] = 0x00; // Lower number
  array_indices_for_display[4] = 0x00; // clear it out first
  updateLEDs=1;
}

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
/* %%%%%%% Functions to test the proper operation of %%%%%%%*/
/* %%%%%%% other code implementations %%%%%%% */
/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

/* Test to ensure we can send data serially to the 8533 device and display it. */
void testLED(void)
{
  // 0001
  sendBit(LOW);
  sendBit(HIGH);
  sendBit(LOW);
  sendBit(HIGH);

  // 0010
  sendBit(LOW);
  sendBit(LOW);
  sendBit(HIGH);
  sendBit(LOW);

  // 0011
  sendBit(LOW);
  sendBit(LOW);
  sendBit(HIGH);
  sendBit(HIGH);
}

/* Tests to make sure random numbers are working. */
void testPrintOutRandomNumbers(void)
{
  printf("random: \%d\n", rand()%16);
}

/* Tests to make sure A/D port is working. */
void testReadADC(void)
{
  unsigned char read = PTIAD & BIT0HI;
  printf(":read \%d\n", read);
  if(read != 0)
  {
    puts("High");
  } else
  {
    puts("Low");
  }
Tests to make timer can properly toggle a bit on off for motor control.

```c
void testTimer(void)
{
    unsigned char expired;
    InitializeTimerRate();
    //setupTimerChannel(TIMER_CHANNEL_0, GATE_TIME);
    while (1)
    {
        expired = checkIsExpired(TIMER_CHANNEL_0);
        printf("expired: %d\r", expired);
        if (expired != 0)
        {
            puts("OFF\r");
            SendMotorCommand (GATE_MOTOR, MOTOR_OFF);
        }
        else
        {
            puts("ON\r");
            SendMotorCommand (GATE_MOTOR, MOTOR_ON);
        }
    }
}
```

Tests to see if the gate motor can be interfaced to a push button & sensor

```c
void testGateButton(void)
{
    unsigned char gateButton;
    unsigned char gateSensor;
    gateButton = debouncedSampleOfPortAD() & BIT4HI; // masking port AD0
    gateSensor = sampleOfPortAD() & BIT7HI;
    if ((gateButton == 0) && (gateSensor == 0)) // press the button but lockout if nothing in slot
    {
        motor_list[GATE_MOTOR].initialize = START_MOTOR_PROCESS; // drive the motor
    }
}
```

Sends out a sequence of bits to ensure 8533 is working and we've wired the LEDs correctly.

```c
void testSerialData(void)
{
    unsigned char twelveBits[12] = {0,1,1,1,0,0,1,1,0,1,0,1};
    sendSerialData(twelveBits);
}