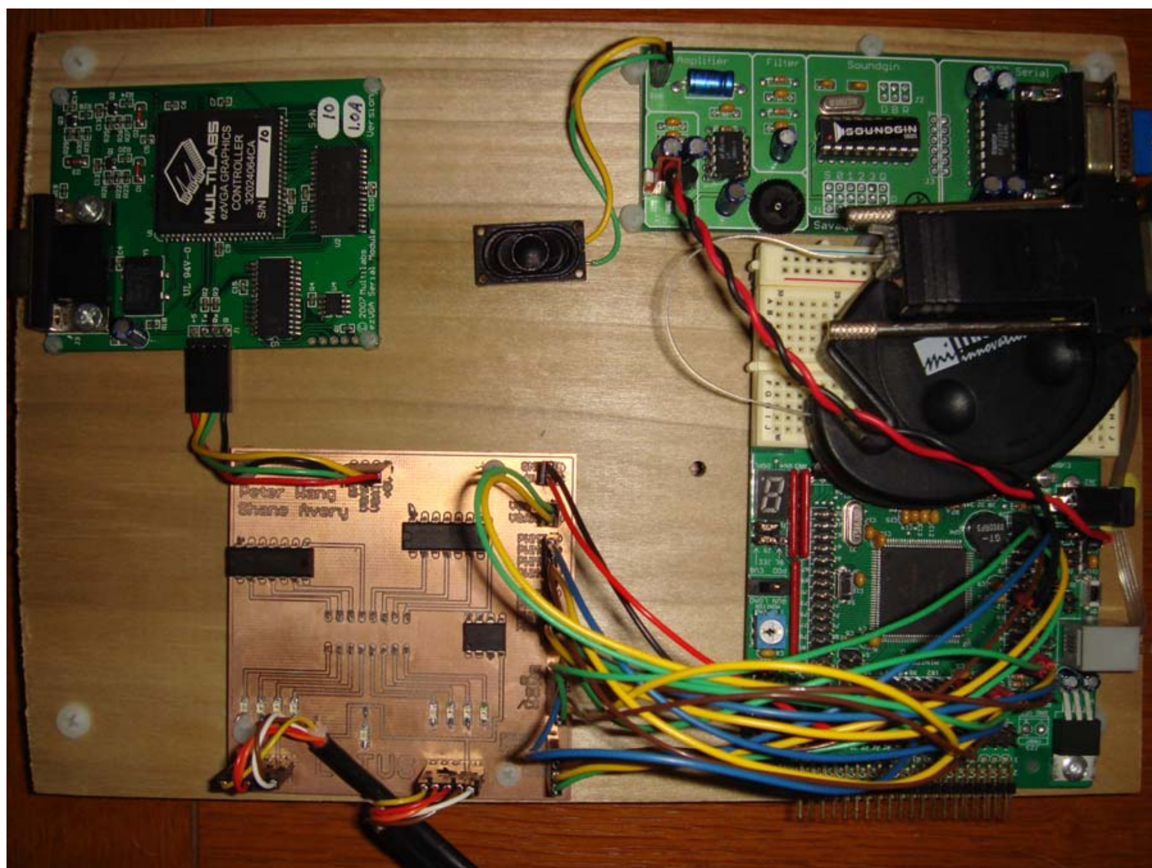
LOTUS: Video Game System

An HCS12 Implementation

By Shane Avery and Peter Wang

Spring 2007 ECE 625



Abstract:

Videogame systems such as Nintendo or Sega had a huge influence on the generation that grew up in the 80's and 90's. The early generation models are perfect examples of a microcontroller application in that they are simple but did its job really well. The Lotus Videogame System builds on the concept of providing an abstraction layer so that to write games, a programmer no longer has to have an intimate knowledge of the hardware. This paper will show the steps involved in creating that level of abstraction in both software and hardware. In addition it will show the creation of two games using the various features the abstraction level provides.

Introduction:

The greatest challenge for a programmer of a video game system is to gain the intimate knowledge of the hardware. This knowledge can be a blessing as the programmer need to know exact details of the machine to maximize the performance needed to provide the player with a unique video game experience. This is known as "squeezing" the performance out of a system. The downside of this however is the time is takes (the ramp-up time) to become intimate with the system to achieve such results. Microsoft attempted to break this trend with the XBOX by creating a video game system that looked very much like a PC. This in affect helped game makers in that they didn't need to learn a new foreign architecture to squeeze performance out of the XBOX.

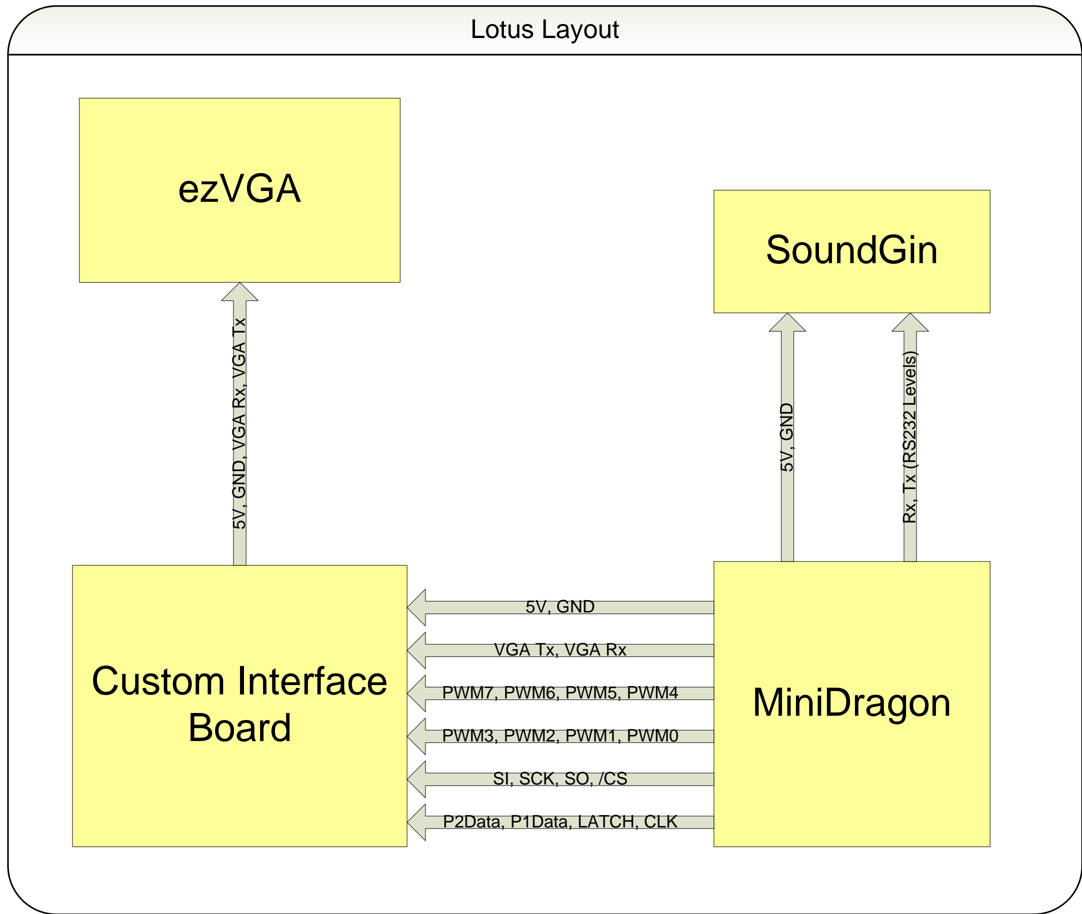
Back in the 8-bit NES era, having an intimate knowledge of the hardware was a good idea because the architecture was simple and the ramp up time was relatively quick. However, the gaming machines will continue to change (in fact Microsoft has left PC architecture for the XBOX360) which forces the game designers to learn a new architectures. The current generation of gaming hardware is so complicated that a new programmer would take years to become familiar with it. By that point, new technology advances will most likely make the current hardware obsolete. The solution to this problem is to create IO libraries which are tuned specifically to a particular game architecture. This allows the game designer to focus on the creation of games and not on learning the hardware. Game development can now leave the effort of creating IO libraries which "squeeze" performance out of an architecture to the people who built the hardware and design games that can be played across all platforms. Having the ability of easily porting a single game to multiple systems is also beneficial since the game will reach a wider audience bringing in additional income.

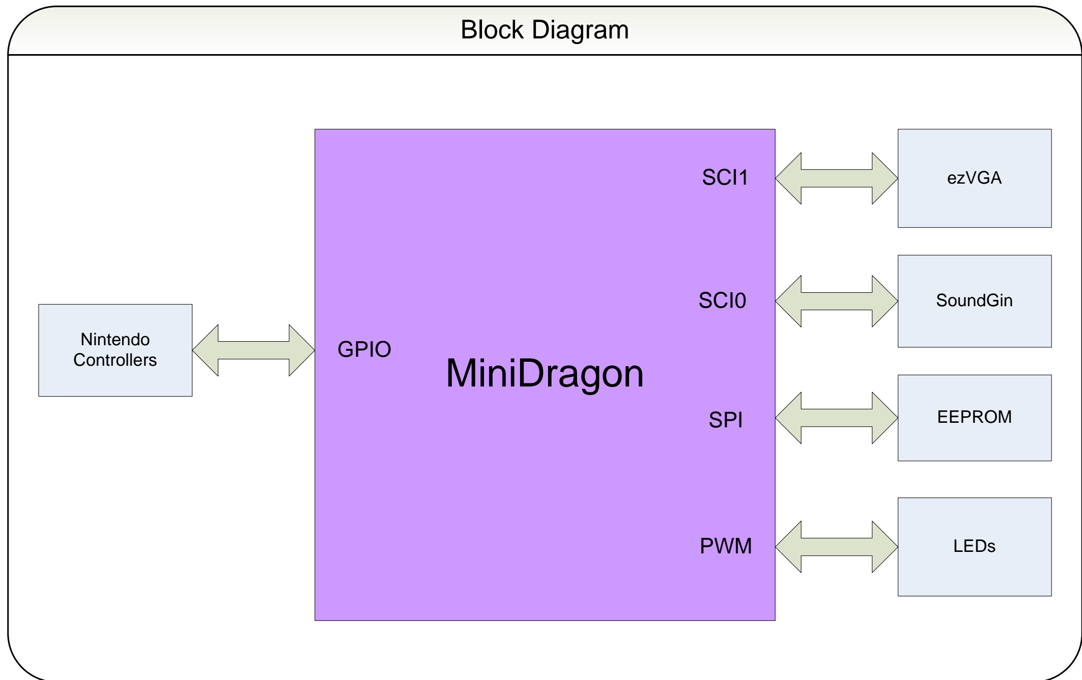
This paper will show how we designed the hardware for the Lotus video game system based off the Motorola HCS12 architecture and how we wrote the software IO libraries which serve as an abstraction layer for game designers. In addition we will show how to use these libraries to create games via two examples.

Lotus Layout and Block Diagram:

Below are diagrams of the Lotus layout and a simple block diagram of the system. The Lotus layout is meant to show how the various hardware components are laid out on the wood base as seen the picture above. Physically the MiniDragon connects to SoundGin via the download cable that was included with the development kit in conjunction with a cross over cable. It should be noted that these are RS232 level signals. All other components of the Lotus system are interfaced to the MiniDragon board via the Custom Interface Board.

The Block Diagram is meant to give a simple look at what peripherals make up the Lotus system as well as what IO modules in the HCS12 are used to communicate with those peripherals.


Lotus Layout
ezVGA
SoundGin
5V, GND, VGA Rx, VGA Tx
5V, GND
Rx, Tx (RS232 Levels)
Custom Interface Board
MiniDragon
5V, GND
VGA Tx, VGA Rx
PWM7, PWM6, PWM5, PWM4
PWM3, PWM2, PWM1, PWM0
SI, SCK, SO, /CS
P2Data, P1Data, LATCH, CLK


Block Diagram
Nintendo Controllers
GPIO
MiniDragon
SCI1
SCI0
SPI
PWM
ezVGA
SoundGin
EEPROM
LEDs

Hardware:

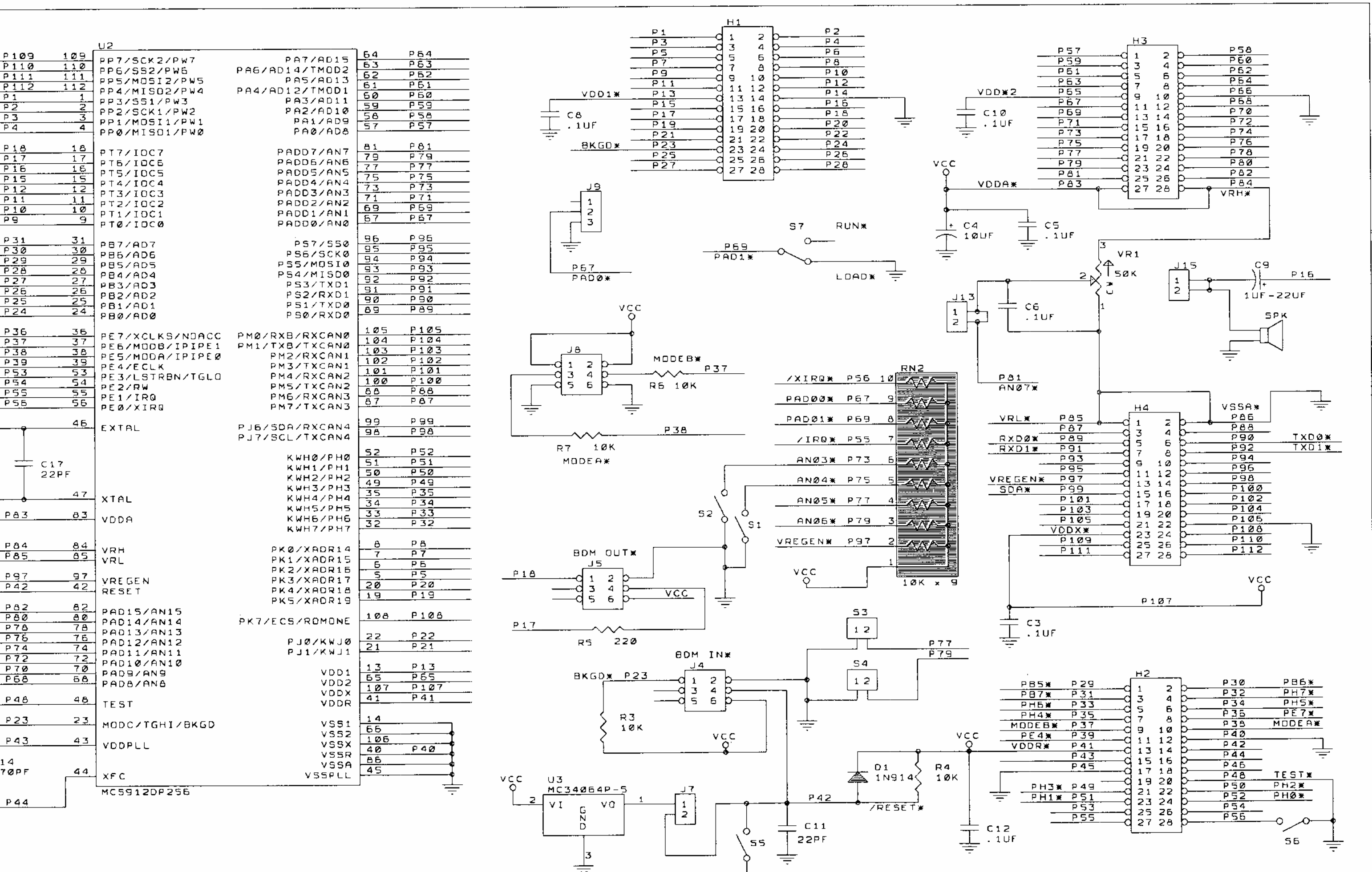
1. 68HCS12

Above is a picture of the MiniDragon+ Development board from www.evbplus.com. This is the board that is used for the Lotus project and it contains a MC9S12DP256. This controller has 256KB of Flash, 12KB of SRAM, 4KB of EEPROM and is internally clocked at 24MHz.

The power supply that came with the development kit didn't provide enough current for the entire Lotus system and thus an 800mA regulated power supply was bought instead. The programming cable was used to program code into the device as first in SRAM and later in Flash. In addition the cable was used for debug and in the end to communicate with the SoundGin device.

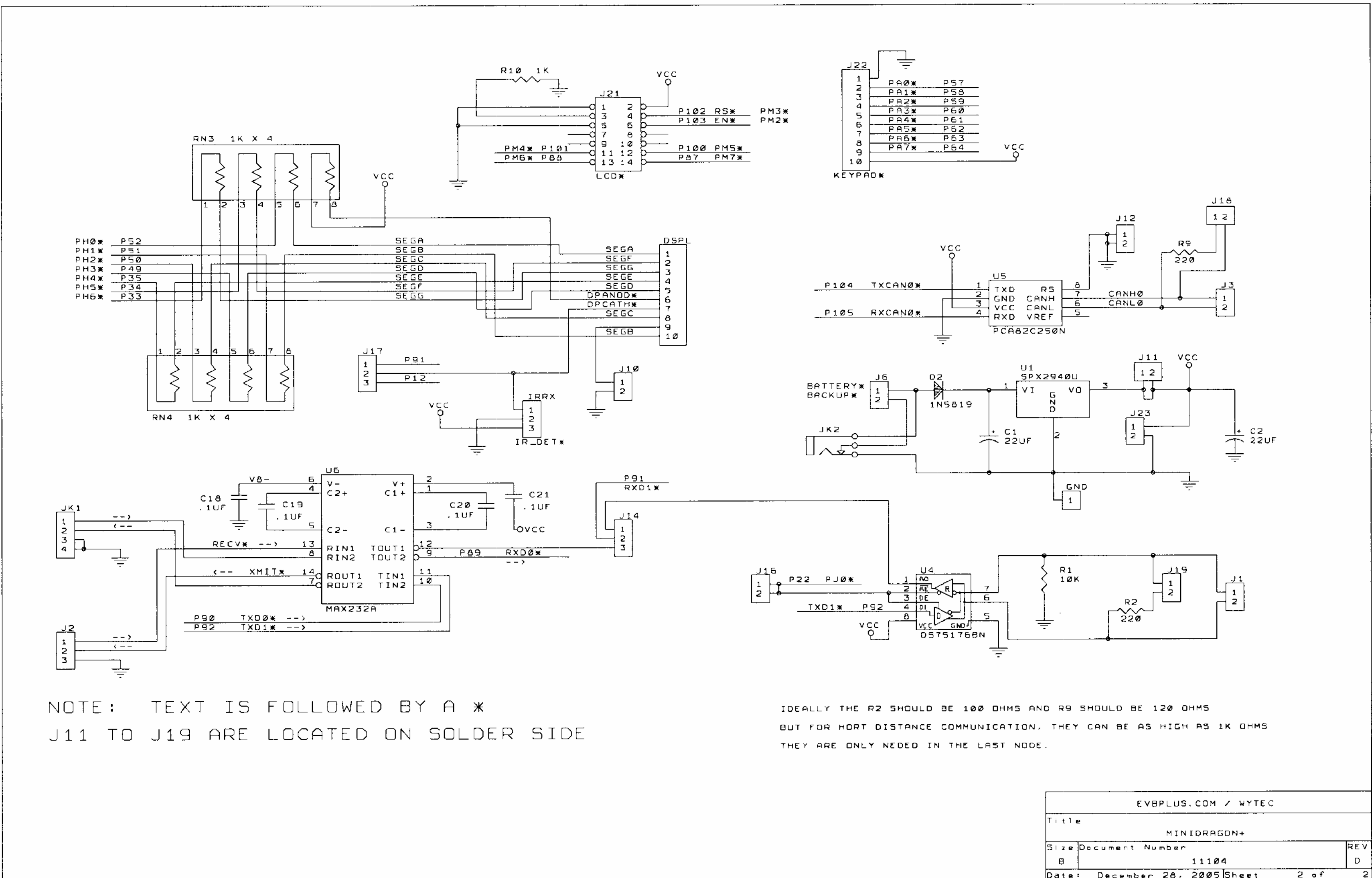
Aside from the actual controller and the download cable the only other thing that Lotus took advantage of in the MiniDragon was the male pin headers. There are four dual row male pin headers that run around the controller that directly connect to every pin of the controller. This was the way that the Custom Interface Board connected to the controller itself. Female pin headers that were crimped to colored wires provided the actual connections from the male pin headers of the Custom Interface Board to the male pin headers of the MiniDragon+.

For reference the schematic for the MiniDragon+ is provided below:



NOTE: TEXT IS FOLLOWED BY A *

EVBPLUS.COM / WYTEC			
Title	MINIDRAGON+		
Size B	Document Number 11104		REV D
Date: December 8, 2005	Sheet 1 of 2		



NOTE: TEXT IS FOLLOWED BY A *
J11 TO J19 ARE LOCATED ON SOLDER SIDE
IDEALLY THE R2 SHOULD BE 100 OHMS AND R9 SHOULD BE 120 OHMS
BUT FOR HORT DISTANCE COMMUNICATION, THEY CAN BE AS HIGH AS 1K OHMS
THEY ARE ONLY NEDED IN THE LAST NODE.
EVBPLUS.COM / WYTEC
Title
MINIDRAGON+
Size B
Document Number 11104
REV D
Date: December 28, 2005
Sheet 2 of 2
RN3 1K X 4
RN4 1K X 4
R10 1K
J21
LCD*
P102 RS*
P103 EN*
PM3*
PM2*
PM4* P101
PM6* P88
P100 PM5*
P87 PM7*
J22
KEYPAD*
PA0* P57
PA1* P58
PA2* P59
PA3* P60
PA4* P61
PA5* P62
PA6* P63
PA7* P64
VCC
PH0* P52
PH1* P51
PH2* P50
PH3* P49
PH4* P35
PH5* P34
PH6* P33
SEGA
SEGB
SEGC
SEGD
SEGE
SEGF
SEGG
DSPL
DPANOD*
DPCATH*
J17
P91
P12
J10
IRRX
IR_DET*
U5
TXD
GND
VCC
RXD
RS
CANH
CANL
VREF
PCA82C250N
P104 TXCAN0*
P105 RXCAN0*
CANH0
CANL0
J12
J18
R9 220
J3
U1
SPX2940U
VI
VO
GND
J11
J6
BATTERY*
BACKUP*
D2
1N5819
C1 22UF
C2 22UF
J23
JK2
GND
U6
MAX232A
V-
C2+
C2-
V+
C1+
C1-
RIN1
RIN2
ROUT1
ROUT2
TOUT1
TOUT2
TIN1
TIN2
V8-
C18 .1UF
C19 .1UF
C20 .1UF
C21 .1UF
JK1
J2
RECV* -->
<-- XMIT*
P89 RXD0*
P91 RXD1*
P90 TXD0* -->
P92 TXD1* -->
J14
J16
P22 PJ0*
TXD1* PS2
U4
DS75176BN
R1 10K
R2 220
J19
J1

2. Video

The video hardware is handled via a dedicated piece of hardware. The device is called ezVGA Serial Module and it is made by Multilabs (http://multilabs.net). The ezVGA Serial Module contains the ezVGA graphics controller, a PIC microcontroller that provides the serial interface, SRAM memory that contains the frame buffer, and an EEPROM that stores the ASCII bitmap. It requires 5 volts for power and consumes about 200mA. The majority of this is due to the Xilinx CPLD that makes up the ezVGA controller. Additional features include BAUD rates from 300 to 115.2K, 320 by 240 resolution, 64 colors, and a floating character command. The four pins header (which is clearly labeled) contains connections for 5V, GND, Serial TX, and Serial RX.

The serial module is accessed via the SCI of the HCS12 at a BAUD rate of 115200. Any command sent the module will be responded with either an ASCII ACK or a NACK. The data sheet contains the commands for the module and they include commands for placing text, defining and placing a floating character, and bitmap operations (such as draw line, draw pixel, etc.). The serial module is physically interfaced to the HCS12 via the custom interface board as can be seen in the picture of Lotus in the title page.

3. Sound

The sound is provided by the SSG01 Sound Coprocessor or SoundGin as referred to in this document. SoundGin is a single chip that contains 6-voice electronic music synthesizer, sound effects, and voice synthesizer. It is capable of producing complex sound effects, synthesizer style music and English speech.

This chip is incredible! It is capable of Amplitude Modulation, Frequency Modulation, Ring Modulation, ADSR Envelopes, and Sound Morphing just to name a few features. If we were to acquire just the chip itself we would need to design a filter and amplifier as well. As a result we purchased the development kit which includes the SoundGin chip, filter, amplifier, and RS232 Level Shifter. The device requires at least 6V to work so we connect SoundGin to directly to the MiniDragon regulated power supply. The SoundGin development kit requires RS232 level serial data which is convenient since the SCI0 interface to the MiniDragon is RS232 level.

The only thing missing from the kit is a speaker which we have to provide ourselves. SoundGin expects to drive an 8ohm speaker and we bought one from Digikey and attached it to the SoundGin development kit speaker header to complete the sound hardware peripheral.

4. Controllers

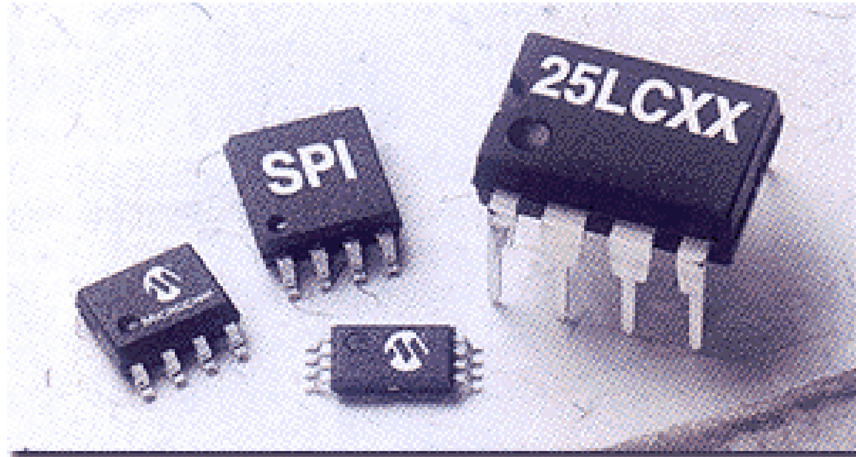
The controllers are the controllers from the Nintendo Entertainment System (NES). Millions of these units were sold in the 80's and continue to be used in other game systems today (such as the Hydra game system). Internally the controller contains only a CMOS 4021 shift register. The form factor connector of the old NES controllers could not be found and thus we had to cut the connectors, strip the wires, and fit them into a female header. Physically they connect to the HCS12 via the Custom Interface Board. The five colored wires are as follows:

White – 5V
Brown – GND
Orange – Latch
Red – Clock
Yellow – Data

The active high latch must be asserted for 200ns. This will latch in the inputs and hold their state. At this point the first piece of data is already available on the Data line. To get the next piece of data we must clock the device on the Clock line. Thus to get all eight inputs we need to clock the device a total of seven times. The order that the data comes out corresponds to the following buttons on the controller:

Data 0 = A
Data 1 = B
Data 2 = Select
Data 3 = Start
Data 4 = Up
Data 5 = Down
Data 6 = Left
Data 7 = Right

5. EEPROM



The EEPROM device is a 25LC160A from Microchip. It is a 16Kbit SPI device with a voltage range from 2.5V-5V and is in a 8 pin DIP package. It is organized as 2K x 8bit and has 16 byte pages. There is an active low write protect line that is pulled high. Also, there is an active low hold pin that is also pulled high. The remaining pins are the SPI interface pins which are routed to the HCS12 pins via the Custom Interface Board.

The first byte sent to the device is one of the following instructions:
1. Read – Read data.
2. Write – Write data.
3. WRDI – Disable write operations.
4. WREN – Enable write operations.
5. RDSR – Read the status register.
6. WRSR – Write status register.

To read data first send the RDSR command to read the status register to be sure that the device is ready to accept a read command. Once the device is ready, send the Read command along with the 16 bit address. The device will then send the byte at that address. To write data first send the RDSR command to read the status register and be sure that device is ready to accept a write command. Once the device is ready, send the WREN command to allow a write to the device. This needs to be done every time you write a byte to the device. After that, send the Write command, 16 bit address, and then the data to write to the device.

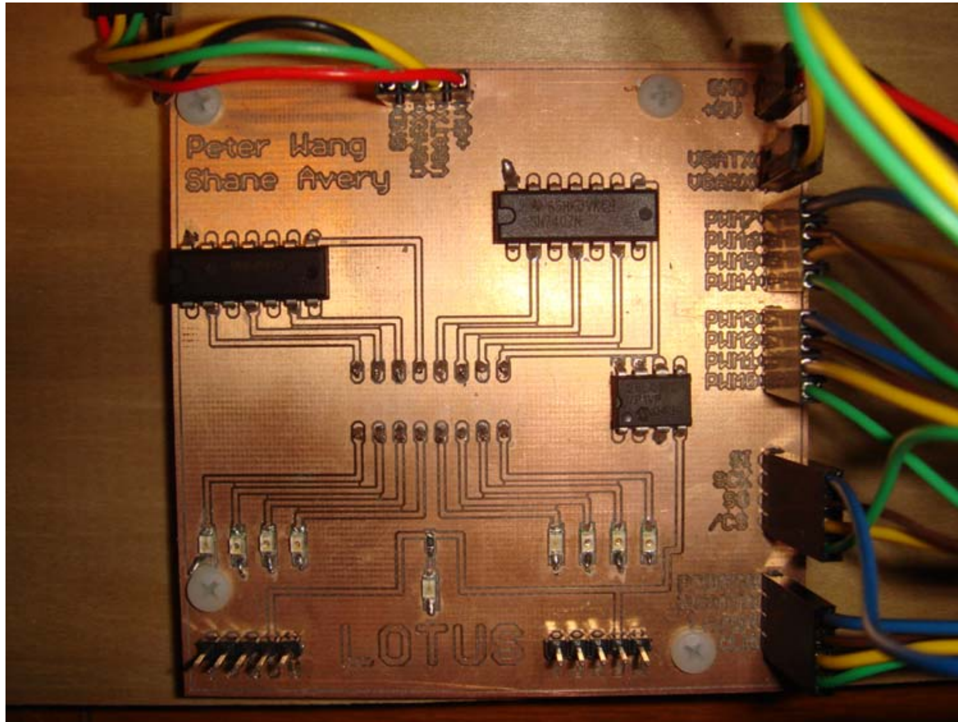
6. LEDs

The LEDs are 0805 package surface mount devices that are soldered to the Custom Interface Board. There is one Blue LED that is a power led that is tied to the power rail that turns on whenever the power is applied to Lotus. There are two Blue LEDs, two Red LEDs, and four Green LEDs. The Custom Interface Board contains eight LEDs that are indirectly tied to the eight PWM outputs of the HCS12. This will give the LEDs the ability to have a “fade” effect (LEDs can be more than just ON or OFF).

The LEDs are not however directly driven by the PWM pins of the HCS12. The PWM pins are connected to two 7407 open collector output buffers ICs which drive the LEDs. The LEDs are current limited by a 220 ohm DIP resistor pack.

The two 7407 open collector ICs are powered by 5V and sink the current that run through the LEDs when they are on. As a result, a logic low on the 7407 input will light the LED. Thus, the PWM unit on the HCS12 should be configured to have an inverted polarity.

7. Custom Interface Board



Above is a picture of the Custom Interface Board. The schematic capture and board layout was done in Eagle. The boards were cut using a Protomat machine that plots the layout to X,Y coordinates and literally drills out the non-copper portions of the PCB. It is a two sided PCB in which the bottom (not shown) contains the bypass capacitors and the DIP resistor pack.

The Custom Interface Board provides an interface from the HCS12 to the ezVGA, EEPROM, LEDs (PWM), and controllers via male pin headers. Connections from the male pin headers of the Custom Interface Board to the MiniDragon board male pin headers (which connect directly to the pins of the HCS12) are listed below:

+5V – Pin 107 VGA Tx – Pin 91
GND – Pin 106 VGA Rx – Pin 92

PWM0 – Pin 4 PWM1 – Pin 3 PWM2 – Pin 2 PWM3 – Pin 1
PWM4 – Pin 112 PWM5 – Pin 111 PWM6 – Pin 110 PWM7 – Pin 109

SI – Pin 94 SCK – Pin 95 SO – Pin 93 /CS – Pin 28

CLK – Pin 24 LATCH – Pin 25 P1DATA – Pin 26 P2DATA – Pin 27

Below are pictures of the Custom Interface Board schematic and board layout.

PWM LEDs

SPI EEPROM

VGA Interface

NES Controllers

Power

Lotus

4/02/2007 08:14:20p

Sheet: 1/1

Peter Wang

Shane Avery

GND VGARX VGATX +5V

GND +5V JP1

VGATX VGARX JP3

PWM7 PWM6 PWM5 PWM4 J2

PWM3 PWM2 PWM1 PWM0 J1

SI SCK SO /CS J7

P2DATA P1DATA LATCH CLK J6

LED1 LED2 LED3 LED4 LED5 LED6 LED7 LED8 LED9

J4 Y O R B W

J5 Y O R B W

LOTUS

8. Parts List

MiniDragon+ - www.evbplus.com - $99.00
ezVGA - http://multilabs.net/ezVGA_SM.html - $64.95
SoundGin - http://thebotshop.com - $49.99
EEPROM – digikey.com - $1.90
Speaker – digikey.com - $7.90
Blue LEDs - digikey.com - $4.98
Green LEDs - digikey.com - $7.50
Red LEDs - digikey.com - $3.00
220 ohm Resistor Pack - digikey.com - $1.20
7404 OC Buffer - digikey.com - $3.84
Female housings - digikey.com - $12.33
Female crimp headers - digikey.com - $9.86
Misc (i.e. wire, shrink wrap, etc.) - $5.00

IO Library:

The IO library is the key to the Lotus system. It provides functions that add a layer of abstraction so that the game designer need not know how to directly communicate with the hardware peripherals of the system. All code was written in C using the GCC compiler. The IDE used was EmbeddedGNU which can be found at www.ericengler.com/EmbeddedGNU.aspx.

1. Initialization

Before the programmer writes any code of his own he should first call the Init() function. This function will initialize all HCS12 peripherals for use with the IO library. Here is a list of the things this function will initialize:

- Initialize the Enhanced Capture Timer for use in seeding the Random Number Generator.
- Initialize PORTB outputs for use in the Controllers.
- Initialize the SPI for use in the EEPROM.
- Initialize the PWM for use in the fade effect of the LEDs.
- Initialize the RTI to give the game designer a sense of time.
- Initialize SCI0 for use with the SoundGin sound chip.
- Initialize SCI1 for use with the ezVGA Serial Module.
- Wait for 300ms for the ezVGA Serial Module to properly come up.
- Send a ‘U’ to the ezVGA Serial Module so that it configures its own serial BAUD rate via Auto BAUD.
- Wait for the ezVGA Serial Module to ACK or NACK its Auto BAUD.

Click on the hyperlink to go to the code for the Init() function: Init()

2. Video

The ezVGA Serial Module has an auto baud feature that'll detect the serial baud rate when starting up. In order to access this, the software must send out a synchronizing command ('U', 0x55) for the ezVGA to lock on to. If successful, the ezVGA will return an ACK signaling that it is ready and everything is fine. In our software this procedure is done in the initialization function (see above).

The ezVGA color scheme is based on mixing the 3 primary colors (red, blue, and green) each at one of 3 intensities. The color is represented as a byte with the lowest 2 bits is red, the next 2 bits is green, and next 2 bits is blue. The 2 most significant bits are not used.

The ezVGA Serial Module has a total of 9 commands and the communication follows a strict structure that always leads off with the actual command byte. All commands will return a value to indicate if everything is working properly. The following is an outline of the more commonly used commands.

-Area Clear Command:
This command clears an area on the screen into any of the 64 colors. The user only has to input the starting and ending X/Y position and the color of the area.

-Background Color Command:
This command set the background color to any of the 64 colors. Any object already on the screen will not be changed.

-Clear Screen Command:
This command clears the entire screen of all objects in to any of the 64 colors.

-Line Command:
This command draws a line between the starting and ending X/Y positions in the color that the user specified.

-Place Character Command:
This command places a defined character on the screen in the user defined color, size, and X/Y position. User is also able to write the character in the regular or inverse style.

-Read and Write Pixel Command:
The Write command changes the specific pixel into the color that the user has defined. The Read command will return the pixel color value of the user defined position instead of an ACK or NAK. The Read command is the only command that does not return an ACK or NAK.

Click on the hyperlink to go to the code for the ezVGA library: ezVGA

3. Sound

The SoundGin chip also requires a very strict structure for commands. The very first byte sent for all commands is the escape character (27 or 0x1B) followed by the command byte and then the parameters for the commands. SoundGin does not return for any command except for the read one byte. The following are some of the commonly used functions.

- WriteOneByteWMask:

We used this command to change the envelope of the sound in our programs by writing to the address of the Oscillator A1's Envelope Decal (Addr 30). The envelope attempts to simulate the way a note is played on a piano or a guitar string. All addresses are defined in SoundGin.h.

- LoadPlayNote:

This command loads a predefined note into a specified oscillator and plays it. All notes are defined in SoundGin.h.

- ReleaseOscillator:

This clear the oscillator so that any sound that's currently play will stop. All oscillators are defined in SoundGin.h.

Click on the hyperlink to go to the code for the SoundGin library: SoundGin

4. Controllers

The only software written for the controller is to shift in the current controller command and make it available. This is done by latching the general purpose IO for 400ns so that the current controller state can be saved. We then released the latch for another 400ns before starting to read it out. The first bit can be read out after the latch is released and the rest of the controller state can be read out during the next 7 clock cycles. Both controllers are read in at the same time to save time. When the function GetContData() is called, the controller inputs are stored in the global value ContP1 and ContP2.

The controllers are setup in the Init() function by setting the general purpose IO PORTB's data flow to have bits 0, 1, and 4 as the output. The output also has the clock and latch as low, and the chip select signal to be high.

Click on the hyperlink to go to the code for the Controller functions: Controller

5. EEPROM

The EEPROM is communicated through the SPI, which does away with sending out a null bit to get the device's attention. The only necessary step needed to access the EEPROM is to set or clear the Chip Select (CS) line. The EEPROM is setup during the Init() function to set the clock rate to 1.5 MHz and changed the HCS12 board as the master. The IO library allows the software to write a byte of data into a specific address and read from any address as well. All addresses are 16-bit values and all data stored are unsigned char.

-Write:
The write function first clears the CS line for the Write Enable command before setting the CS line. When the status returns as ready, the write command will require the CS line to be cleared again for the actual write command followed by the 16 bit address and then the data before setting the CS line. The function will then block and check in on the EEPROM until it returns with a status showing that it's done. The following is an example of how to write the value 14 into the address 300.

```
WriteEEPROM(300, 14);
```

-Read:
The read function first clears the CS line for the write command and then the 16 bit address. The software will then blocks until the data is returned before setting the CS. The read value is returned from the function. The following is an example of how to read the value stored in address 300.

```
unsigned char value = ReadEEPROM(300);
```

Click on the hyperlink to go to the code for the EEPROM functions: Read and Write

6. LEDs

The PWM duty cycle library to affect the LED's fade effect required the PWM polarity to be set on active low making the LED glow brighter with a high duty cycle. The PWM clock is also divided by 32 because the LED controls do not need a very fast clock. In the fade effect function, an input is required to enter a value from 0 to 7 for the LED address. On the Lotus board, the left most LED above the controller input is the LED number 0 and counts up. The power indication LED is not addressed. The second input required is the intensity of the LED with 0 as off and 255 as the brightest. The following is an example of how to set the third LED from the left to a quarter of max intensity.

```
SetPWMDuty(2, 64);
```

Click on the hyperlink to go to the code for the LEDs PWM function: LED

7. Random Number Generator

The IO library uses the enhanced capture timer to seed the Random Number Generator. The timer is started in the Init() function and from then on runs continuously. As some point the game designer needs to call the Seed() function which will use the value that is in the ECT as that time as a seed for the Random Number Generator. The game designer should take great care not to make the Seed() function call at a deterministic time as the number will then be the same every time. It is advised to use player input to determine when the Random Number Generator is seeded.

Once the Seed() function has been called any call to the Rand() function will result in a new random number. The Rand() function is simply a 16bit LFSR with taps at 16, 15, 13, and 4. The Rand() function will return the new random number as an unsigned int and in addition a global variable called RandNum will be assigned the new random number.

8. RTI

The Real Time Interrupt is set in the Init() function to fire off an interrupt at approximately 30Hz and flag a global value. During the main function of the code, this flag is constantly polled for the 30Hz tick. Each program written for the Lotus must have all operations finished before the next 30Hz tick to avoid timing issues. If the Interrupt happens before all of the code are finished executing, the missed interrupt will lead to errors in the program executing because parts of the code will be operating on a different time frame. Controlled blocking is permitted if all operations needs longer than 30Hz to complete by waiting for the next interrupt to occur and clear the flag. Below is a typical setup for the main loop that will wait until the RTI fires before running its code.

```
// Main loop.
while(1)
{
    // Wait for the RTI flag to be set.
    if(RTIFlag)
    {
        RTIFlag = 0;    // First thing we do is clear the flag.

        . . .

    }
}
```

Below is an example of how to make the controlled blocking call within the main program executing.

```
    if(RTIFlag)
    {
        RTIFlag = 0;    // First thing we do is clear the flag.

        . . .

        while(RTIFlag == 0);    // Wait for RTIFlag to be set.
        RTIFlag = 0;            // Clear it.

        . . .

    }
```

Click on the hyperlink to go to the code for the Real Time Interrupt function: RTI

Example Games:

1. Tron Bike Knockoff

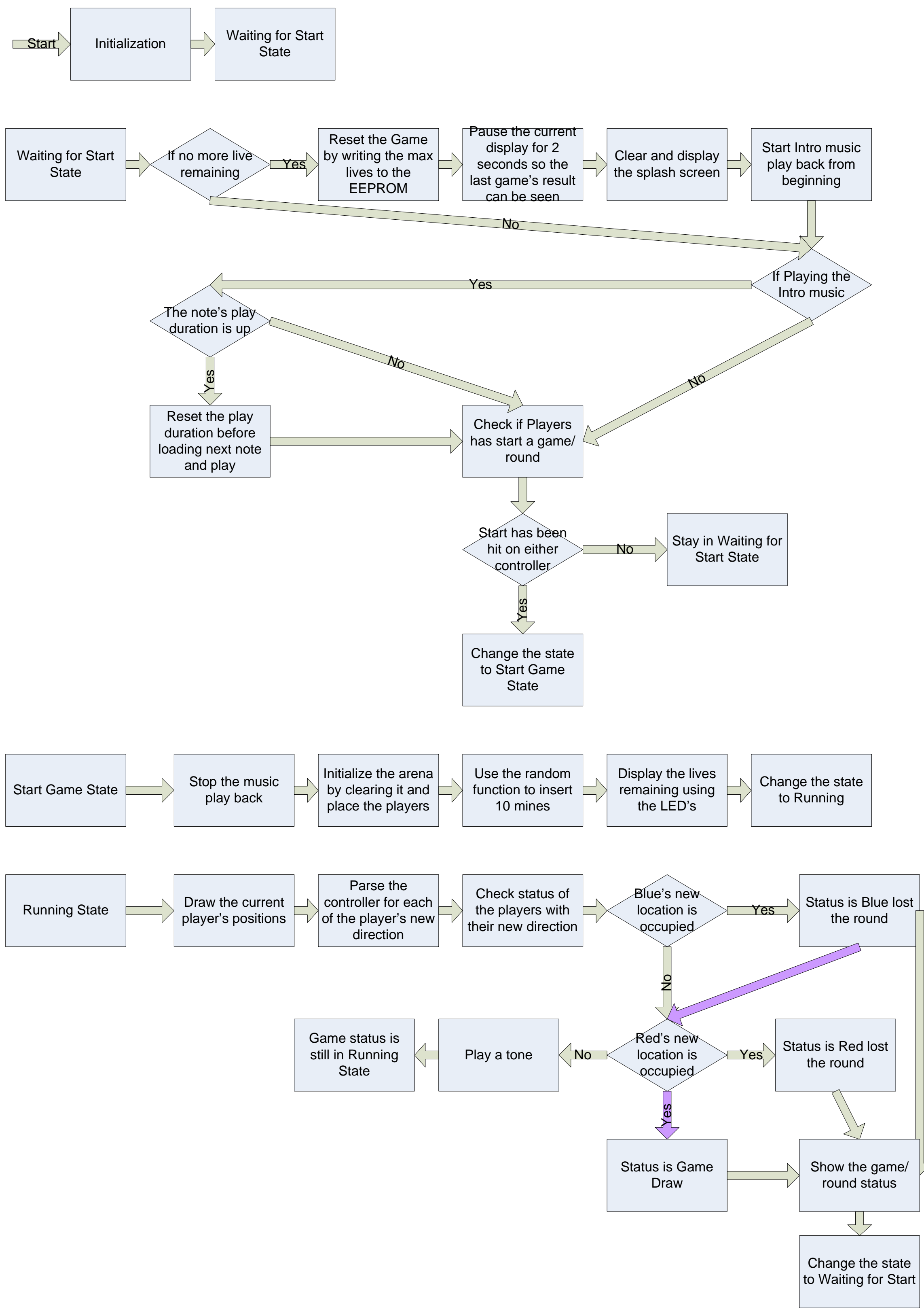
The Tron Bike game pits two players (red and blue) against each other in a closed off arena. Each player is in constant motion and a wall erects in their trail. The objective of the game is to knock off the other player by forcing them to run into the arena wall or the trail walls. The arena is also dotted with mines that the players can use to their advantage. Each player has 4 lives that's displayed on the custom interface board and the game ends when one player loses all of their lives. This game also features a splash screen before the game begins and an intro music that's played on loop. During the game, the player can move in all directions and each round will continue until one of the players gets knocked off. A draw condition happens when both players gets knocked off at the same time.

The game is implemented as a 3 state machine: Waiting for Start, Start Game, and Running. All three states are executed well before the 30Hz time limit is up. A controlled block is used during the Waiting for Start state to hold the current screen for 2 second.

Waiting for Start:
In this state, the game checks the controller to see if any of the players had pushed the start button. For a new game, the current screen is held for 2 seconds before the splash screen is drawn and the intro music is played. The intro music's notes and corresponding play duration has already been predefined and two local index keeps track of which note to play and how much longer to play that note. For a game in progress, nothing new is displayed.

Start Game:
Once either player has hit the start button, this state will clear the screen and draw out the arena and stop the intro music. The random number generator is also used to set a number of mines. The remaining lives are read from the EEPROM and displayed on the LED bank. This state immediately jumps into the Running state.

Running:
During the Running state, the controller is checked to see if the players had changed to a new direction and move the dots on the screen appropriately. The new location is then checked to see if it has already been occupied by a mine, trail wall, or the arena wall. If the new location is already occupied, then the player has lost the round. Both players are checked for either a win condition or a draw condition. A different noise is played for the draw condition, a win condition, and the game in progress condition. When the round is over, the current victor screen is drawn and the state returns to Waiting for Start.



Start
Initialization
Waiting for Start State
Waiting for Start State
If no more live remaining
Yes
Reset the Game by writing the max lives to the EEPROM
Pause the current display for 2 seconds so the last game's result can be seen
Clear and display the splash screen
Start Intro music play back from beginning
No
If Playing the Intro music
Yes
The note's play duration is up
No
Yes
No
Reset the play duration before loading next note and play
Check if Players has start a game/ round
Start has been hit on either controller
No
Stay in Waiting for Start State
Yes
Change the state to Start Game State
Start Game State
Stop the music play back
Initialize the arena by clearing it and place the players
Use the random function to insert 10 mines
Display the lives remaining using the LED's
Change the state to Running
Running State
Draw the current player's positions
Parse the controller for each of the player's new direction
Check status of the players with their new direction
Blue's new location is occupied
Yes
Status is Blue lost the round
No
Game status is still in Running State
Play a tone
No
Red's new location is occupied
Yes
Status is Red lost the round
Yes
Status is Game Draw
Show the game/ round status
Change the state to Waiting for Start

2. Kill Kevin

Kill Kevin is a reaction based game in which two players race to complete 50 sequenced movements. The random number generator will determine whether the players should press up, down, left, right, A, or B. Players will have to complete the same 50 movements and any mistake will result in the player being penalized by going back 10 movements.

When the player first turns on the game they are greeted with a splash screen. When the user hits a button is when the Random Number Generator is seeded by the ECT and the 50 movements are determined. The players are then given countdown to the start of the game and when the count gets down to zero the game begins. When a player makes a mistake not only do they get knocked down by 10 but the players hear an annoying noise to indicate the error. The closer a player gets to 50 the more "intense" the game music and the more the LEDs light up. When one player reaches 50 then game ends and plays the winning music.

At any time the players can press start or select to get to the Save/Restore menu. Here players can save the current game to EEPROM or restore a previous game from EEPROM.

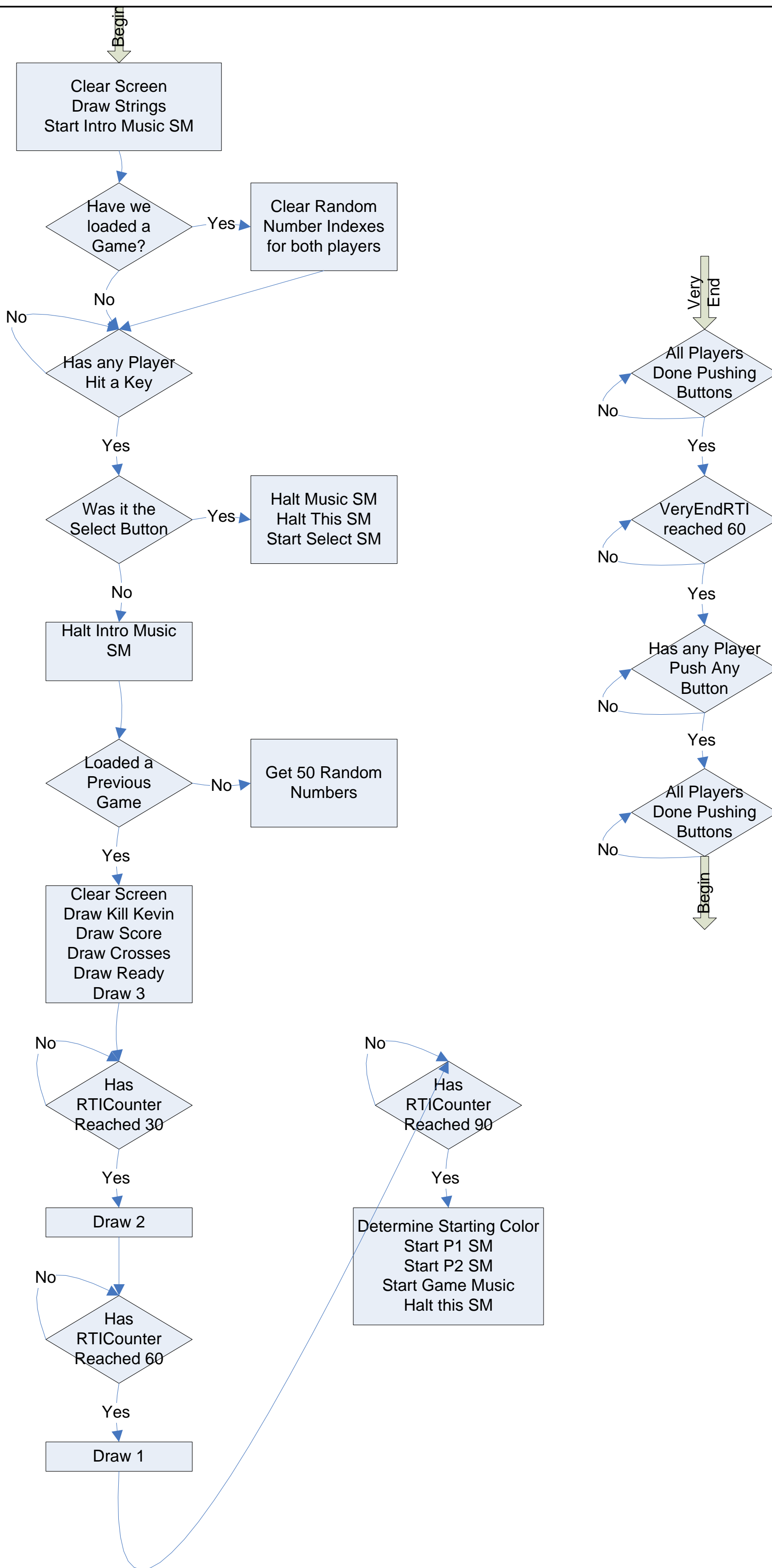
The game is made up of five state machines. A brief description of the state machines is below followed by a flow diagram of each state machine.

SplashSM: This SM displays the Splash Screen and starts the Intro Music State Machine. It simply waits for the user to press any button to begin the game. When the user does, it generates the 50 Random Numbers and starts the Player State Machines.

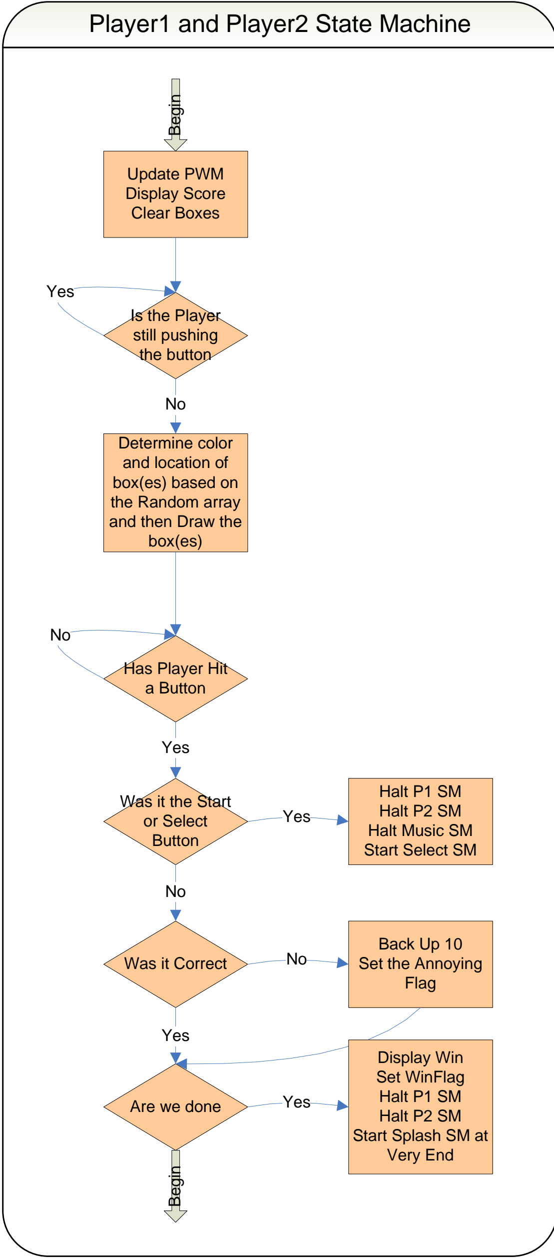
Player SM: There are two player state machines. Essentially they are identical and run independently of each other. The SM first updates and displays the score and the LEDs. It then determines based on the random number array what to display next for the player and then waits for player input. Once the player hits a button the SM determines whether it was correct or not. If wrong, it tells the Music SM to play the annoying noise. We then determine if we have reached 50 and if so we play the winning music.

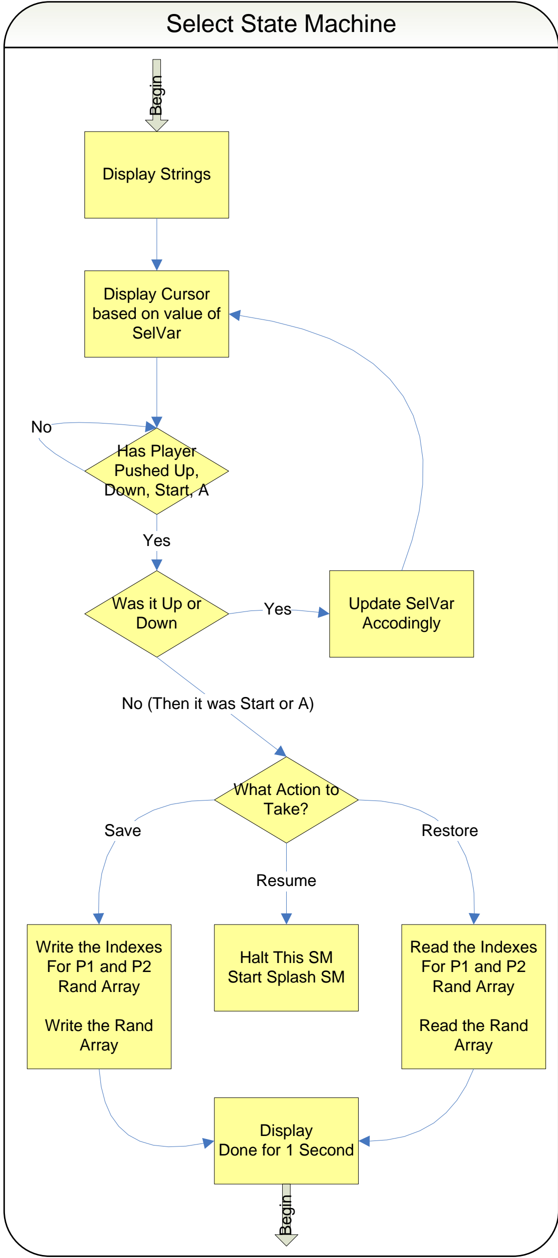
Music SM: The Music SM begins my playing the intro music which is from 2001 Space Odyssey. When game play begins the Music SM just plays the same two notes over and over again but plays them faster and faster depending on how close the players are to 50. When a player makes a mistake the Music SM will play an annoying sound to indicate the error. After a player has reached 50 the Music SM will play the winning music which is "We Are the Champions."

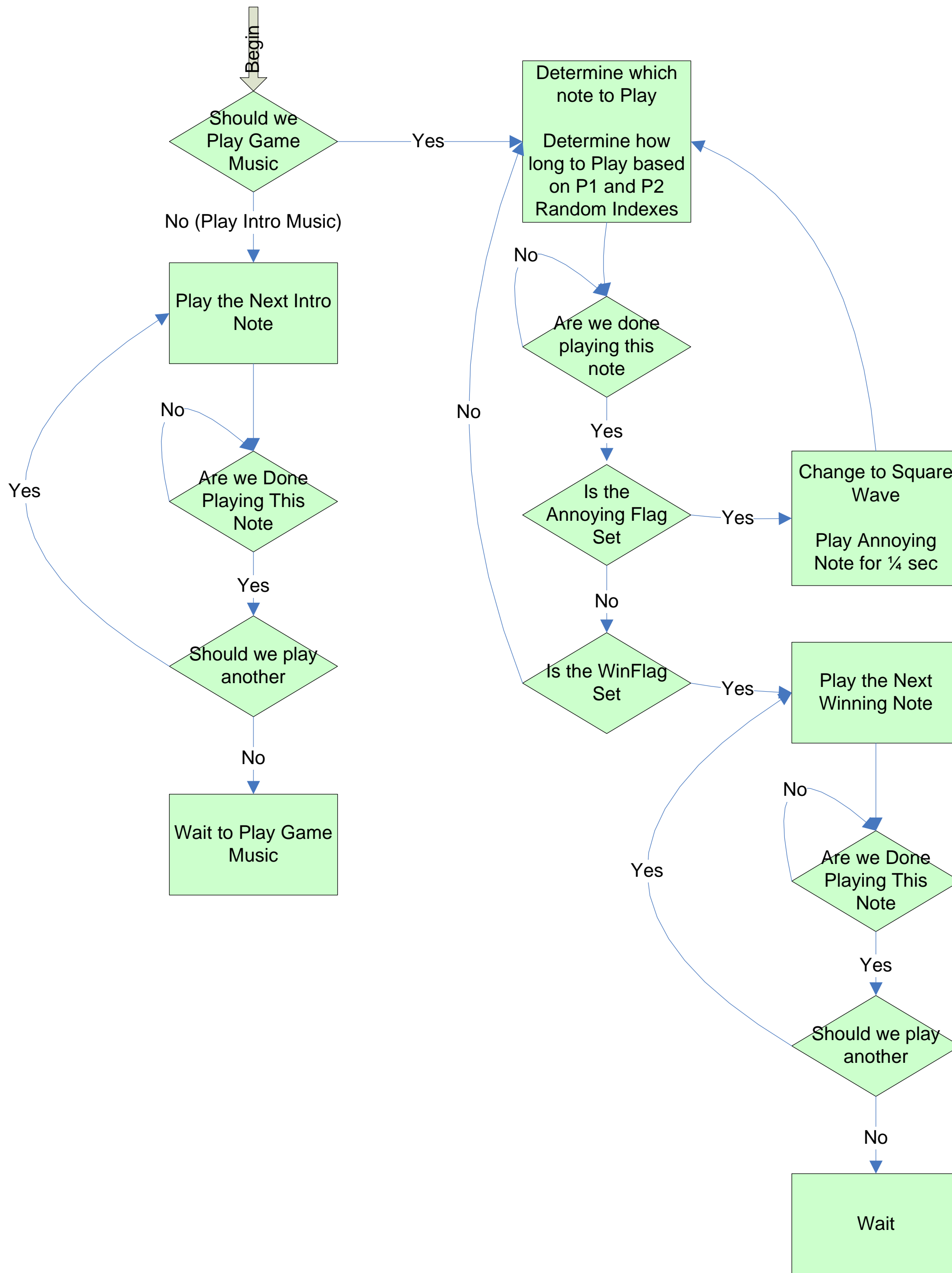
Select SM: The Select SM is started when one of the players hits the start or select button. This will display a menu for the players to save or restore a game.



Splash State Machine
Begin
Clear Screen
Draw Strings
Start Intro Music SM
Have we loaded a Game?
Yes
Clear Random Number Indexes for both players
No
No
Has any Player Hit a Key
Yes
Was it the Select Button
Yes
Halt Music SM
Halt This SM
Start Select SM
No
Halt Intro Music SM
Loaded a Previous Game
No
Get 50 Random Numbers
Yes
Clear Screen
Draw Kill Kevin
Draw Score
Draw Crosses
Draw Ready
Draw 3
No
Has RTICounter Reached 30
Yes
Draw 2
No
Has RTICounter Reached 60
Yes
Draw 1
No
Has RTICounter Reached 90
Yes
Determine Starting Color
Start P1 SM
Start P2 SM
Start Game Music
Halt this SM
Very End
All Players Done Pushing Buttons
No
Yes
VeryEndRTI reached 60
No
Yes
Has any Player Push Any Button
No
Yes
All Players Done Pushing Buttons
No
Begin


Player1 and Player2 State Machine
Begin
Update PWM
Display Score
Clear Boxes
Yes
Is the Player still pushing the button
No
Determine color and location of box(es) based on the Random array and then Draw the box(es)
No
Has Player Hit a Button
Yes
Was it the Start or Select Button
Yes
Halt P1 SM
Halt P2 SM
Halt Music SM
Start Select SM
No
Was it Correct
No
Back Up 10
Set the Annoying Flag
Yes
Are we done
Yes
Display Win
Set WinFlag
Halt P1 SM
Halt P2 SM
Start Splash SM at Very End
Begin


Select State Machine
Begin
Display Strings
Display Cursor based on value of SelVar
No
Has Player Pushed Up, Down, Start, A
Yes
Was it Up or Down
Yes
Update SelVar Accodingly
No (Then it was Start or A)
What Action to Take?
Save
Resume
Restore
Write the Indexes For P1 and P2 Rand Array
Write the Rand Array
Halt This SM
Start Splash SM
Read the Indexes For P1 and P2 Rand Array
Read the Rand Array
Display Done for 1 Second
Begin


Music State Machine
Begin
Should we Play Game Music
Yes
No (Play Intro Music)
Play the Next Intro Note
No
Are we Done Playing This Note
Yes
Should we play another
Yes
No
Wait to Play Game Music
Determine which note to Play
Determine how long to Play based on P1 and P2 Random Indexes
No
Are we done playing this note
Yes
Is the Annoying Flag Set
Yes
Change to Square Wave
Play Annoying Note for ¼ sec
No
No
Is the WinFlag Set
Yes
Play the Next Winning Note
No
Are we Done Playing This Note
Yes
Should we play another
Yes
No
Wait

Future Work:

1. Video

Add functions to create custom characters and floating characters. Floating characters are a special low flicker character that is only moved during the sync. This could be a mouse pointer for example. Functions are needed not only to define the floating character but for moving it as well.

More complex draw methods are possible using existing library such as draw figure, or draw box.

2. Sound

The SoundGin is capable of doing synthetic voice and this would be a great addition. Another possible future work is to simplify the SoundGin library further so that it's more intuitive to use without an intimate knowledge of the SoundGin hardware. We would also like to see the way SoundGin's command implementation changed to the ezVGA's implementation to take up less memory space.

3. 68HCS12

Currently the largest game can only be the size of 1 flash page. It would advantageous to learn how to program the entire flash for larger, more complex games. This would also allow multiple games to be stored onto the memory.

4. Additional Games

Survivor - A game where two players are faced off against an invading army of foes. Each armed with a ranged and a melee weapon, they must survive to become stronger and faster. An experience system will allow the players to become more powerful as they fight more and the EEPROM could be used to store all players' stats.

Conclusion:

Keeping up with the complexity of the current hardware available is quickly becoming a near impossible task without dramatically increasing the cost of software development. By providing an abstraction layer between the game programmers and the hardware, this can greatly reduced the development cost by cutting the effort that new hardware training requires. The Lotus video game system took this abstraction layer approach to shield the game developers from having to learn how to interface with the hardware such as the HCS12 or the ezVGA.

This paper shows how game designers can go from a software flow diagram and realize their game on a video game system without having to interface directly to the hardware peripherals. For example, all mention of the EEPROM interface in the flow diagrams involves read from or write to, there is very little mention of SPI timing to communicate with the EEPROM. The same trend of having a transparent layer to the hardware is shown in the game specific code. Bulk of the code is logic for the games themselves with very little being for the hardware peripherals of the video games system.

The two games mentioned in this paper were written in approximately one week. If the game programmers had to write these games without the use of the IO library it could have easily taken ten times that long. The elimination of the overhead from having to interface with the hardware resulted in the production of fun games in a very short period of time because the effort is focused only on the game.

A hidden benefit is that a standard has already been established and making the code easier to read and debug. The trade off is that without direct control of the hardware, the general purposed functions can cause a slight performance impact. However, the over all benefits of being standardized really out weights the slight hit in performance.

In conclusion we have shown the reason that video game produces would want to create IO libraries and how they benefit game programmers. We have also shown by example how video game designer would design their system to support this paradigm by writing the IO library code. As a result the future of game design will likely follow this trend to stay competitive.

References:

[1] Steven Barrett and Daniel Pack, Embedded Systems Design and Application with the 68HC12 and 68HCS12, Pearson Prentice Hall, 2005

[2] Steven Barrett, Real Time Embedded Systems Spring 2005, University of Wyoming, Department of Electrical and Computer Engineering, 2005

[3] Andre LaMothe, Game Programming for the Propeller Powered Hydra, Parallax Inc, 2006

[4] Ashley Geng, ECE 625 Microprocessor Application Notes, CSUN, Department of Electrical and Computer Engineering, 2007

Appendix:

1. Library Code

Initialization Code:

```
// Initalizing all of the registers on the dragon board for the lotus system
void Init()
{
    char index;

    // Initialize the counter used for seeding the Random Number generator.
    TSCR1 = 0x80;

    // Initialze the Controller and SPI CS signal.
    DDRB = 0x13;      // Make bits 0,1, and 4 output.
    PORTB = 0x10;     // Make clk low and latch low.
                      // Make SPI CS signal high.

    // Initialize the SPI for the EEPROM.
    SPI0BR = 0x03;    // Set the SPI clk rate to 1.5MHz.
    SPI0CR1 = 0x50;   // Turn on SPI and make us a Master.

    // Initialize the PWM.
    PWMPOL = 0;       // Active low polarity. This means the greater the duty
                      // cycle then brighter the LED.

    PWMPRCLK = 0x55;  // This makes clock to the PWM divided by 32 which is
                      // good because the LED can't switch that fast.

    PWME = 0xFF;       // Enable all PWM channels.

    // Turn the LEDs off.
    for(index=0;index<8;index++)
        SetPWMDuty(index, 0x00);

    // Initialize the RTI.
    CRGINT |= RTIE;   // Enable the interrupt for the RTI.
    RTICTL = 0x78;    // Set the RTI frequency to 30.5Hz.

    // Initialize SCIs for SoundGin and VGA.
    // Initialize SCI0
    SCI0BDH = 0;      // br=MCLK/(16*baudRate)
    SCI0BDL = 0x9C;   // 9600 BAUD.
    SCI0CR1 = 0;
    /* bit value meaning
    7   0    LOOPS, no looping, normal
    6   0    WOMS, normal high/low outputs
    5   0    RSRC, not appliable with LOOPS=0
    4   0    M, 1 start, 8 data, 1 stop
    3   0    WAKE, wake by idle (not applicable)
    2   0    ILT, short idle time (not applicable)
    1   0    PE, no parity
    0   0    PT, parity type (not applicable with PE=0) */
    SCI0CR2 = 0x0C;
    /* bit value meaning
    7   0    TIE, no transmit interrupts on TDRE
    6   0    TCIE, no transmit interrupts on TC
    5   0    RIE, no receive interrupts on RDRF
    4   0    ILIE, no interrupts on idle
    3   1    TE, enable transmitter
    2   1    RE, enable receiver
    1   0    RWU, no receiver wakeup
    0   0    SBK, no send break */

    // Initialize SCI1
    SCI1BDH = 0;      // br=MCLK/(16*baudRate)
    //SCI1BDL = 13;     // 115200 BAUD.
    SCI1BDL = 14;     // 115200 BAUD.
    SCI1CR1 = 0;
    /* bit value meaning
    7   0    LOOPS, no looping, normal
    6   0    WOMS, normal high/low outputs
    5   0    RSRC, not appliable with LOOPS=0
    4   0    M, 1 start, 8 data, 1 stop
    3   0    WAKE, wake by idle (not applicable)
    2   0    ILT, short idle time (not applicable)
    1   0    PE, no parity
    0   0    PT, parity type (not applicable with PE=0) */
    SCI1CR2 = 0x0C;
    /* bit value meaning
    7   0    TIE, no transmit interrupts on TDRE
    6   0    TCIE, no transmit interrupts on TC
    5   0    RIE, no receive interrupts on RDRF
    4   0    ILIE, no interrupts on idle
    3   1    TE, enable transmitter
    2   1    RE, enable receiver
    1   0    RWU, no receiver wakeup
    0   0    SBK, no send break */

    // Wait for about 300ms for ezVGA to initialize.
```

```
    for(index=0;index<10;index++)
    {
        while(RTIFlag == 0);    // Wait for RTIFlag to be set.
        RTIFlag = 0;            // Clear it.
    }

    // Send the U character to ezVGA so that it know the BAUD rate.
    while((SCI1SR1 & TDRE) == 0);
    SCI1DRL = 'U';

    // Wait for ACK/NACK. This guarantees that we wont send
    // ezVGA another byte until he is ready.
    while((SCI1SR1 & RDRF) == 0);
    if(SCI1DRL == 0x15)       // Check to see if it is a NACK.
    {
        SetPWMDuty(1, 0xff);  // Turn on both red LEDs to indicate a problem.
        SetPWMDuty(5, 0xff);

        // Wait for about 1s for user to see there is a problem.
        for(index=0;index<30;index++)
        {
            while(RTIFlag == 0);    // Wait for RTIFlag to be set.
            RTIFlag = 0;            // Clear it.
        }

        SetPWMDuty(1, 0x00);  // Turn off both red LEDs.
        SetPWMDuty(5, 0x00);
    }

    /* Any user initializations here. */
}
```

RTI ISR:

```
// Interrupt Handler routine for RTI.
void __attribute__((interrupt)) RTIHandle()
{
    CRGFLG |= RTIF;   // Clear the interrupt flag.
    RTIFlag = 1;      // Set the global variable flag.

    /* Any user code needed withing the ISR here. */
}
```

PWM Duty Cycle (LED Fade Effect):

```
// The pulse width modulator function controls 8 LEDs on the Lotus Board.
// The available 8 LEDs are referrenced from 0-7 with 0 being the left most
// LED and 7 being the right most.  The intensity of the LED can be set in
// a range between 0 and 255 with 255 being the brightest.
void SetPWMDuty(char PWMChannel, unsigned char Value)
{
    switch(PWMChannel)
    {
        case 0:    PWMDTY0 = Value;
                   break;

        case 1:    PWMDTY1 = Value;
                   break;

        case 2:    PWMDTY2 = Value;
                   break;

        case 3:    PWMDTY3 = Value;
                   break;

        case 4:    PWMDTY4 = Value;
                   break;

        case 5:    PWMDTY5 = Value;
                   break;

        case 6:    PWMDTY6 = Value;
                   break;

        case 7:    PWMDTY7 = Value;
                   break;
    }
}
```

Get Controller Data:

```
unsigned char ContP1;
unsigned char ContP2;

// This Controller function reads out the shift registers and store in two
// public values called ContP1 and ContP2.  ContP1 is the left controller
// and ContP2 is the right one.
void GetContData()
{
    char index;
```

```
    ContP1 = 0;
    ContP2 = 0;

    // Be sure that we start out with clk and latch set to zero.
    PORTB &= 0xfc;

    // Shift in data from controllers.
    // First we latch it.
    PORTB |= 2;

    // Wait 400ns
    __asm__ __volatile__  (" nop ");__asm__ __volatile__  (" nop ");
    __asm__ __volatile__  (" nop ");__asm__ __volatile__  (" nop ");
    __asm__ __volatile__  (" nop ");__asm__ __volatile__  (" nop ");
    __asm__ __volatile__  (" nop ");__asm__ __volatile__  (" nop ");
    __asm__ __volatile__  (" nop ");__asm__ __volatile__  (" nop ");

    // Release the latch
    PORTB &= 0xfd;

    // Wait 400ns
    __asm__ __volatile__  (" nop ");__asm__ __volatile__  (" nop ");
    __asm__ __volatile__  (" nop ");__asm__ __volatile__  (" nop ");
    __asm__ __volatile__  (" nop ");__asm__ __volatile__  (" nop ");
    __asm__ __volatile__  (" nop ");__asm__ __volatile__  (" nop ");
    __asm__ __volatile__  (" nop ");__asm__ __volatile__  (" nop ");

    // Get the data.
    if(PORTB & 0x04)
        ContP1 |= 0x01;

    if(PORTB & 0x08)
        ContP2 |= 0x01;

    for(index = 0;index < 7;index++)
    {
        // Bring the clock high.
        PORTB |= 0x01;

        ContP1 = ContP1 << 1;
        ContP2 = ContP2 << 1;

        // Bring the clock low.
        PORTB &= 0xfe;

        if(PORTB & 0x04)
            ContP1 |= 0x01;

        if(PORTB & 0x08)
            ContP2 |= 0x01;
    }

}
```

Random Number Generation:

```
unsigned int RandNum;

// This is a library function that will seed the random number generator
// with the timer/counter value.
void Seed()
{
    RandNum = TCNT;
}

// This is a library function which will return a pseudo random number.
// Note that the user should first call the Seed() function one time before
// calling Rand(). After that the user can call Rand() over and over again to
// get a random number. Note that the user can also access the random number
// at any time as it is a global variable.
unsigned int Rand()
{
    unsigned int temp;
    temp = ( RandNum << 1 ) ^ ((( RandNum & 0x8000 ) >> 15) |         /* 16 tap */
                               (( RandNum & 0x8000 ))       |         /* 15 tap */
                               (( RandNum & 0x8000 ) >> 2)  |         /* 13 tap */
                               (( RandNum & 0x8000 ) >> 11));         /* 4 tap */

    RandNum = temp;
    return temp;
}
```

EEPROM Write:

```
// Note that when writing EEPROM we will block until the operation is complete.
// Thus, it is not advisable to do EEPROM write during gameplay as it will
// violate the RTI.
void WriteEEPROM(unsigned int add, unsigned char data)
{
    unsigned char MSB = add >> 8;
    unsigned char LSB = add;
    unsigned char dummy;

    // First we need to enable the write.
    PORTB &= 0xef;   // Clear CS.
    SPI0DR = 0x06;   // Send Command to enable writes.
    while((SPI0SR & SPIF) == 0);  // Wait for us to write data.
    PORTB |= 0x10;   // Set CS.
    dummy = SPI0DR;  // Need to access SPI0DR to clear the interrupt.

    // Now write the data.
    PORTB &= 0xef;   // Clear CS.
    SPI0DR = 0x02;   // Send Command to write.
    while((SPI0SR & SPIF) == 0);  // Wait for us to write data.
    dummy = SPI0DR;  // Need to access SPI0DR to clear the interrupt.

    SPI0DR = MSB;    // Write the MSB of the address.
    while((SPI0SR & SPIF) == 0);  // Wait for us to write data.
    dummy = SPI0DR;  // Need to access SPI0DR to clear the interrupt.

    SPI0DR = LSB;    // Write the LSB of the address.
    while((SPI0SR & SPIF) == 0);  // Wait for us to write data.
    dummy = SPI0DR;  // Need to access SPI0DR to clear the interrupt.

    SPI0DR = data;    // Write the data.
    while((SPI0SR & SPIF) == 0);  // Wait for us to write data.
    dummy = SPI0DR;  // Need to access SPI0DR to clear the interrupt.
    PORTB |= 0x10;   // Set CS.

    // Now wait for the data to be written.
    dummy = 0xff;
    while(dummy & 0x01)
    {
        PORTB &= 0xef;     // Clear CS.
        SPI0DR = 0x05;     // Send the command to read the status register.
        while((SPI0SR & SPIF) == 0); // Wait for us to write command.
        dummy = SPI0DR;    // Dummy read to clear the interrupt.

        SPI0DR = 0xff;     // Dummy write value to clock data in.
        while((SPI0SR & SPIF) == 0); // Wait for dummy write to complete.
        dummy = SPI0DR;    // Dummy value we care about.
        PORTB |= 0x10;     // Set CS
    }

}
```

EEPROM Read:

```
// Note that when reading EEPROM we will block until the operation is complete.
// Thus, it is not advisable to do EEPROM read during gameplay as it will
// violate the RTI.
unsigned char ReadEEPROM(unsigned int add)
{
    unsigned char dummy;
    unsigned char MSB = add >> 8;
    unsigned char LSB = add;
    unsigned char retval;

    // Now write the data.
    PORTB &= 0xef;   // Clear CS.
    SPI0DR = 0x03;   // Send Command to read.
    while((SPI0SR & SPIF) == 0);  // Wait for us to write data.
    dummy = SPI0DR;  // Need to access SPI0DR to clear the interrupt.

    SPI0DR = MSB;    // Write the MSB of the address.
    while((SPI0SR & SPIF) == 0);  // Wait for us to write data.
    dummy = SPI0DR;  // Need to access SPI0DR to clear the interrupt.

    SPI0DR = LSB;    // Write the LSB of the address.
    while((SPI0SR & SPIF) == 0);  // Wait for us to write data.
    dummy = SPI0DR;  // Need to access SPI0DR to clear the interrupt.

    SPI0DR = 0xff;   // Dummy write to get the data.
    while((SPI0SR & SPIF) == 0);  // Wait for us to write data.
    retval = SPI0DR; // Need to access read the data we wrote.
    PORTB |= 0x10;   // Set CS.

    return retval;

}
```

ezVGA Serial Module Functions:

```
unsigned int ezVGABufferSize = 0;
char ezVGABuffer[10];

// Issue command
char ezVGAIssueCMD()
{
    int index = 0;
    volatile int temp;

    // Write until all command has been written
    for(index=0;index<ezVGABufferSize;index++)
    {
        while((SCI1SR1 & TDRE) == 0) {};
        SCI1DRL = ezVGABuffer[index];
    }

    // Wait for ACK/NACK. This guarantees that we wont send
    // ezVGA another byte until he is ready.
    while((SCI1SR1 & RDRF) == 0);   // Wait for data.
    temp = SCI1DRL;                 // Need to read register for flag to clear.

    return 0;
}

// This command can be used in a few different ways.  It can be used to clear a
// specific area on the screen or it can be used to draw filled squares and
// rectangles.  Each area is defined by two points.  The upper left-hand corner
// is referred to as point x1,y1 and the lower right-hand point as x2,y2.  When
// this command is executed the ezVGA Serial Module will draw an area from
// point x1,y1 to x2,y2.  If the color is the same as the background color then
// it will appear that the area was erased.  If any other color is used a
// square/rectangle will appear on the screen.
char ClearArea(char color, int x1, char y1, int x2, char y2)
{
    // Assemble the x values into high and low bits
    char x1HighByte = x1 >> 8;
    char x1LowByte = (char)x1;

    char x2HighByte = x2 >> 8;
    char x2LowByte = (char)x2;

    ezVGABuffer[0] = V_AREACLEAR;
    ezVGABuffer[1] = color;
    ezVGABuffer[2] = x1HighByte;
    ezVGABuffer[3] = x1LowByte;
    ezVGABuffer[4] = y1;
    ezVGABuffer[5] = x2HighByte;
    ezVGABuffer[6] = x2LowByte;
    ezVGABuffer[7] = y2;

    ezVGABufferSize = 8;

    // Issue command
    return(ezVGAIssueCMD());
}

// This command is used to change the background color of the screen.  Any
// objects on the screen and the floating character are not affected by this
// command, they are drawn around.
char SetBackground(char color)
{
    ezVGABuffer[0] = V_SETBCKGND;
    ezVGABuffer[1] = color;

    ezVGABufferSize = 2;

    // Issue command
    return(ezVGAIssueCMD());
}

// This command is used to clear the entire screen.  The screen is cleared to
// the color specified and this becomes the new background color.  The floating
// character is automatically turned off when this command is executed.  If it
// is needed it must be turned back on after the command is completed.
char ClearScreen(char color)
{
    ezVGABuffer[0] = V_SCRNCLEAR;
    ezVGABuffer[1] = color;

    ezVGABufferSize = 2;

    // Issue command
    return(ezVGAIssueCMD());
}

// Create a Character.  This command allows you to define your own custom
// characters or change any of the ASCII characters that came with the ezVGA
// Serial Module.  The character memory map is setup to allow 256 total
// characters.  Characters index 0 through 31 are undefined, 32 through 127
// are pre-defined from the factory to a standard ASCII character, and 128 to
// 255 are undefined.  To create a custom character you first draw the image
// that you want in an 8 by 8 matrix (as shown above).  Once you have what you
// want you have to determine a value for each of the eight rows.  At the
```

```
// bottom of each column there is a number starting at 128 under the left-most
// column.  These numbers represent the value of each pixel that is turned on
// in that row.  To get the byte value for a row you add up all the values for
// each pixel that is turned on.
char CreateCharacter(char idx, char row0, char row1, char row2, char row3,
                     char row4, char row5, char row6, char row7)
{
    ezVGABuffer[0] = V_CREATCHAR;
    ezVGABuffer[1] = idx;
    ezVGABuffer[2] = row0;
    ezVGABuffer[3] = row1;
    ezVGABuffer[4] = row2;
    ezVGABuffer[5] = row3;
    ezVGABuffer[6] = row4;
    ezVGABuffer[7] = row5;
    ezVGABuffer[8] = row6;
    ezVGABuffer[9] = row7;

    ezVGABufferSize = 10;

    // Issue command
    return(ezVGAIssueCMD());
}

// This command enables you to draw a character that will "float" over the
// other characters on the screen without erasing or disturbing them in any way.
char FloatingChar(char idx, char mode, char color, int x, char y)
{
    // Assemble the x values into high and low bits
    char xHighByte = x >> 8;
    char xLowByte = (char)x;

    ezVGABuffer[0] = V_FLOATCHAR;
    ezVGABuffer[1] = idx;
    ezVGABuffer[2] = mode;
    ezVGABuffer[3] = color;
    ezVGABuffer[4] = xHighByte;
    ezVGABuffer[5] = xLowByte;
    ezVGABuffer[6] = y;

    ezVGABufferSize = 7;

    // Issue command
    return(ezVGAIssueCMD());
}

// This command enables you to draw a line on the screen.  Any type of straight
// and angled line can be drawn.  Each line is defined by two points.  The
// first is the starting point which is referred to as x1,y1.  The second is
// the ending point which is x2,y2.  A line can be erased from the screen by
// re-drawing it in the current background color.
char DrawLine(char color, int x1, char y1, int x2, char y2)
{
    // Assemble the x values into high and low bits
    char x1HighByte = x1 >> 8;
    char x1LowByte = (char)x1;

    char x2HighByte = x2 >> 8;
    char x2LowByte = (char)x2;

    ezVGABuffer[0] = V_DRAWALINE;
    ezVGABuffer[1] = color;
    ezVGABuffer[2] = x1HighByte;
    ezVGABuffer[3] = x1LowByte;
    ezVGABuffer[4] = y1;
    ezVGABuffer[5] = x2HighByte;
    ezVGABuffer[6] = x2LowByte;
    ezVGABuffer[7] = y2;

    ezVGABufferSize = 8;

    // Issue command
    return(ezVGAIssueCMD());
}

// This command places a character on the screen.  The character can be placed
// in a variety of different ways depending on your needs.  The first option is
// characters can be placed as an opaque or as a transparent character.  When a
// character is placed as an opaque all the pixels are drawn.  The pixels that
// are off are drawn in the current background color and the pixels that are on
// are drawn in the character color.  When drawn this way a character will
// completely overwrite whatever it is placed over.  When a character is placed
// as a transparent then the pixels that are off are not drawn.  Only the
// pixels that are on are drawn in the character color.  This allows you to
// place a character over a background and the background will be visible
// through the off pixels of the character.  Secondly, characters can be placed
// in normal or reverse mode.
char PlaceCharacter(char idx, char mode, char size, char color, int x, char y)
{
    // Assemble the x values into high and low bits
    char xHighByte = x >> 8;
    char xLowByte = (char)x;

    ezVGABuffer[0] = V_PLACECHAR;
    ezVGABuffer[1] = idx;
```

```
    ezVGABuffer[2] = mode;
    ezVGABuffer[3] = size;
    ezVGABuffer[4] = color;
    ezVGABuffer[5] = xHighByte;
    ezVGABuffer[6] = xLowByte;
    ezVGABuffer[7] = y;

    ezVGABufferSize = 8;

    // Issue command
    return(ezVGAIssueCMD());
}

// This command is used to read a pixel from the screen.  When executed the
// color of the pixel at the x,y position is read from the video memory and
// returned as a single byte.
char ReadPixel(int x, char y)
{
    // Assemble the x values into high and low bits
    char xHighByte = x >> 8;
    char xLowByte = (char)x;

    ezVGABuffer[0] = V_READPIXEL;
    ezVGABuffer[1] = xHighByte;
    ezVGABuffer[2] = xLowByte;
    ezVGABuffer[3] = y;

    ezVGABufferSize = 4;

    // Issue command
    return(ezVGAIssueCMD());
}

//This command is used to write a pixel to the screen.
char WritePixel(char color, int x, char y)
{
    // Assemble the x values into high and low bits
    char xHighByte = x >> 8;
    char xLowByte = (char)x;

    ezVGABuffer[0] = V_WRITPIXEL;
    ezVGABuffer[1] = color;
    ezVGABuffer[2] = xHighByte;
    ezVGABuffer[3] = xLowByte;
    ezVGABuffer[4] = y;

    ezVGABufferSize = 5;

    // Issue command
    return(ezVGAIssueCMD());
}

// Mix colors
char MixColor(char Blue, char Green, char Red)
{

    // Limit the color intensity to max
    if(Blue > V_MAXINTENSITY)
    {
        Blue = V_MAXINTENSITY;
    }

    if(Green > V_MAXINTENSITY)
    {
        Green = V_MAXINTENSITY;
    }

    if(Red > V_MAXINTENSITY)
    {
        Red = V_MAXINTENSITY;
    }

    return(V_BLUE*Blue + V_GREEN*Green + V_RED*Red);

}
```

SoundGin Functions:

```
unsigned int SoundBufferSize = 0;
char SoundBuffer[6];

//
// Assemble the command and send it out to the serial port
char IssueCommand(unsigned char Para, unsigned char Cmd, unsigned char Arg1,
                 unsigned char Arg2, unsigned char Arg3, unsigned char Arg4) {

    char index = 0;

    // Set the global setting for the serial send interrupt
    SoundBuffer[0] = S_CMD;
    SoundBuffer[1] = Cmd;
    SoundBuffer[2] = Arg1;
    SoundBuffer[3] = Arg2;
    SoundBuffer[4] = Arg3;
    SoundBuffer[5] = Arg4;

    SoundBufferSize = Para + 2;     // accound for cmd and the issue cmd

    for(index=0;index<SoundBufferSize;index++)
    {
        while((SCI0SR1 & TDRE) == 0){};
        SCI0DRL = SoundBuffer[index];
    }

    return 0;
}

//
// Reads the memory at the location specified by Reg.
// i.e.: 27,0,24 - Causes Oscillator A1's Amplitude to be sent out TX.
char ReadOneByte(unsigned char Reg) {
    return(IssueCommand(1, S_R1B, Reg, 0, 0, 0));
}

//
// Writes Byte at memory at the location specified by Reg.
// i.e.: 27,1,24,127 - Sets Oscillator A1's Amplitude to 127
char WriteOneByte(unsigned char Reg, unsigned char Byte) {
    return(IssueCommand(2, S_W1B, Reg, Byte, 0, 0));
}

//
// Writes two Bytes at memory at the location specified by Reg.
// i.e.: 27,1,20,163,1 - Sets Oscillator A1's Frequency Target to 100Hz
char WriteTwoByte(unsigned char Reg, unsigned char Byte1, unsigned Byte2) {
    return(IssueCommand(3, S_W2B, Reg, Byte1, Byte2, 0));
}

//
// Writes three Bytes at memory at the location specified by Reg.
// i.e.: 27,1,17,110,163,1 - Sets Oscillator A1's Frequency to 100.00Hz
char WriteThreeByte(unsigned char Reg, unsigned char Byte1,
                   unsigned char Byte2, unsigned char Byte3) {
    return(IssueCommand(4, S_W3B, Reg, Byte1, Byte2, Byte3));
}

//
// Writes Byte masked with Mask at memory at the location specified by Reg.
// i.e.:27,4,0,2,255 - Causes Oscillator A2 to be included in Mixer A's output.
char WriteOneByteWMask(unsigned char Reg, unsigned Byte, unsigned char Mask) {
    return(IssueCommand(3, S_W1BM, Reg, Byte, Mask, 0));
}

//
// Clears memory locations 0-127.
char ClearMixerAB() {
    return(IssueCommand(0, S_CMAB, 0, 0, 0, 0));
}

//
// Causes the oscillators to calculate a ramp value and then turn on
// the ramp and target options.
char RampToTargets(char A, char B) {
    if(A & B) {                 // ramp both mixers
        return(IssueCommand(0, S_RTAB, 0, 0, 0, 0));
    } else if(A) {              // ramp only mixer A
        return(IssueCommand(0, S_RTA, 0, 0, 0, 0));
    } else {                    // ramp only mixer B
        return(IssueCommand(0, S_RTB, 0, 0, 0, 0));
    }
}

//
// Sets the Voice frequency to the specified musical note frequency.
char SetVoiceNote(unsigned char Note) {
    return(IssueCommand(1, S_SVN, Note, 0, 0, 0));
}

//
// Sets the Voice frequency to the specified musical note frequency.
char SetVoiceFreq(unsigned int Freq) {
```

```
    // break apart the freq to be stored in 2 chars
    unsigned char lower = (unsigned char) Freq;
    unsigned char upper = (unsigned char) (Freq >> 8);

    return(IssueCommand(2, S_SVF, upper, lower, 0, 0));
}

//
// Sets the Voice delay to the specified 8-Bit amount.
char SetVoiceDelay(unsigned char Delay) {
    return(IssueCommand(1, S_SVD, Delay, 0, 0, 0));
}

//
// Sets the Voice options to the default values.
char SetVoiceDefault() {
    return(IssueCommand(0, S_SVDEF, 0, 0, 0, 0));
}

//
// Turns the Q line on.
char SetQOn() {
    return(IssueCommand(0, S_QON, 0, 0, 0, 0));
}

//
// Turns the Q line off.
char SetQOff() {
    return(IssueCommand(0, S_QOFF, 0, 0, 0, 0));
}

//
// Clears memory locations for mixer and its associated oscillators.
char ClearMixerOscillators(char A) {
    if(A) {                     // clear mixer A
        return(IssueCommand(0, S_CMOA, 0, 0, 0, 0));
    } else {                    // clear mixer B
        return(IssueCommand(0, S_CMOB, 0, 0, 0, 0));
    }
}

//
// Sets the oscillator's frequency to the specified musical note frequency.
char Load1Note(char Osc, unsigned char Note) {
    switch(Osc) {
        case EO_A1: return(IssueCommand(1, S_LNOA1, Note, 0, 0, 0));
        case EO_A2: return(IssueCommand(1, S_LNOA2, Note, 0, 0, 0));
        case EO_A3: return(IssueCommand(1, S_LNOA3, Note, 0, 0, 0));
        case EO_B1: return(IssueCommand(1, S_LNOB1, Note, 0, 0, 0));
        case EO_B2: return(IssueCommand(1, S_LNOB2, Note, 0, 0, 0));
        case EO_B3: return(IssueCommand(1, S_LNOB3, Note, 0, 0, 0));
    }
}

//
// Sets the mixer and its associated oscillators to a predefined sound.
// 27,75,3 - Set the Oscillators in Mixer A to sound 3.
char LoadSound(char A, unsigned char Sound) {
    if(A) {                     // load sound into mixer A
        return(IssueCommand(1, S_LSMA, Sound, 0, 0, 0));
    } else {                    // load sound into mixer B
        return(IssueCommand(1, S_LSMB, Sound, 0, 0, 0));
    }
}

//
// Sets the oscillator's frequency to the specified 16-bit frequency.
char LoadFreq(char Osc, unsigned int Freq) {
    // break apart the freq to be stored in 2 chars
    unsigned char lower = (unsigned char) Freq;
    unsigned char upper = (unsigned char) (Freq >> 8);

    switch(Osc) {
        case EO_A1: return(IssueCommand(2, S_LFOA1, upper, lower, 0, 0));
        case EO_A2: return(IssueCommand(2, S_LFOA2, upper, lower, 0, 0));
        case EO_A3: return(IssueCommand(2, S_LFOA3, upper, lower, 0, 0));
        case EO_B1: return(IssueCommand(2, S_LFOB1, upper, lower, 0, 0));
        case EO_B2: return(IssueCommand(2, S_LFOB2, upper, lower, 0, 0));
        case EO_B3: return(IssueCommand(2, S_LFOB3, upper, lower, 0, 0));
    }
}

//
// Play the oscillator in its current settings.
char PlayCurrentOscillator(char Osc) {
    switch(Osc) {
        case EO_A1: return(IssueCommand(0, S_PCOA1, 0, 0, 0, 0));
        case EO_A2: return(IssueCommand(0, S_PCOA2, 0, 0, 0, 0));
        case EO_A3: return(IssueCommand(0, S_PCOA3, 0, 0, 0, 0));
        case EO_B1: return(IssueCommand(0, S_PCOB1, 0, 0, 0, 0));
        case EO_B2: return(IssueCommand(0, S_PCOB2, 0, 0, 0, 0));
        case EO_B3: return(IssueCommand(0, S_PCOB3, 0, 0, 0, 0));
    }
}
```

```
//
// Play the mixer in its current settings.
char PlayCurrentMixer(char A) {
    if(A) {                    // play mixer A
        return(IssueCommand(0, S_PCMA, 0, 0, 0, 0));
    } else {                   // play mixer B
        return(IssueCommand(0, S_PCMB, 0, 0, 0, 0));
    }
}

//
// Release the oscillator of its current settings.
char ReleaseOscillator(char Osc) {
    switch(Osc) {
        case EO_A1: return(IssueCommand(0, S_ROA1, 0, 0, 0, 0));
        case EO_A2: return(IssueCommand(0, S_ROA2, 0, 0, 0, 0));
        case EO_A3: return(IssueCommand(0, S_ROA3, 0, 0, 0, 0));
        case EO_B1: return(IssueCommand(0, S_ROB1, 0, 0, 0, 0));
        case EO_B2: return(IssueCommand(0, S_ROB2, 0, 0, 0, 0));
        case EO_B3: return(IssueCommand(0, S_ROB3, 0, 0, 0, 0));
    }
}

//
// Release the mixer of its current settings.
char ReleaseMixer(char A) {
    if(A) {                    // release mixer A
        return(IssueCommand(0, S_RMA, 0, 0, 0, 0));
    }
    else {                     // release mixer B
        return(IssueCommand(0, S_RMB, 0, 0, 0, 0));
    }
}

//
// Sets the oscillator's frequency to the specified musical note
// frequency and start the envelope.
// 27,88,64 - Sets Oscillator A1's frequency to C4 (Middle-C) and
// starts the envelope
char LoadPlayNote(char Osc, unsigned char Note) {
    switch(Osc) {
        case EO_A1: return(IssueCommand(1, S_LPNOA1, Note, 0, 0, 0));
        case EO_A2: return(IssueCommand(1, S_LPNOA2, Note, 0, 0, 0));
        case EO_A3: return(IssueCommand(1, S_LPNOA3, Note, 0, 0, 0));
        case EO_B1: return(IssueCommand(1, S_LPNOB1, Note, 0, 0, 0));
        case EO_B2: return(IssueCommand(1, S_LPNOB2, Note, 0, 0, 0));
        case EO_B3: return(IssueCommand(1, S_LPNOB3, Note, 0, 0, 0));
    }
}

//
// Sets the oscillator's frequency to the specified 16-bit frequency
// and starts the envelope.
char LoadPlayFreq(char Osc, unsigned int Freq) {
    // break apart the freq to be stored in 2 chars
    unsigned char lower = (unsigned char) Freq;
    unsigned char upper = (unsigned char) (Freq >> 8); // Got it pete.

    switch(Osc) {
        case EO_A1: return(IssueCommand(2, S_LPFOA1, upper, lower, 0, 0));
        case EO_A2: return(IssueCommand(2, S_LPFOA2, upper, lower, 0, 0));
        case EO_A3: return(IssueCommand(2, S_LPFOA3, upper, lower, 0, 0));
        case EO_B1: return(IssueCommand(2, S_LPFOB1, upper, lower, 0, 0));
        case EO_B2: return(IssueCommand(2, S_LPFOB2, upper, lower, 0, 0));
        case EO_B3: return(IssueCommand(2, S_LPFOB3, upper, lower, 0, 0));
    }
}

//
// Reset to voice parameters.
char ResetVoiceParameter(char A) {
    if(A) {                    // reset voice A
        return(IssueCommand(0, S_RVPA, 0, 0, 0, 0));
    } else {                   // reset voice B
        return(IssueCommand(0, S_RVPB, 0, 0, 0, 0));
    }
}
```

2. Tron Bike Knockoff Specific Code

```
// Blue variables
unsigned int m_Blue_PosX;
unsigned char m_Blue_PosY;
char m_Blue_Dir;
char m_Blue_Color;
char m_Blue_Life;

// Red variables
unsigned int m_Red_PosX;
unsigned char m_Red_PosY;
char m_Red_Dir;
char m_Red_Color;
char m_Red_Life;

// Main Function
int main()
{
    // Declare splash screen text
    char BWString[9] = "BLUE WINS";
    char RWString[8] = "RED WINS";
    char TieString[9] = "GAME DRAW";
    char TitleString[20] = "Tron Bike: Knockoff!";
    char PSString[18] = "Push Start to Play";

    // Splash screen music
    unsigned char SCMusic[12] = {E3, E3, E3, C3, E3, G3, G3, C3, G3, E3, A3, B3};
    unsigned char SCMTime[12] = {10, 20, 20, 10, 20, 20, 20, 30, 20, 30, 20, 30};
    char MTimeLeft = 0;
    char MIndex = 0;
    char MStart = 0;

    // Setup the colors
    char Status = WFSTART;

    // General purpose for loop index counter
    int index;

    // First call the Init function to intialize all the IO.
    Init();

    // Write to the EEPROM max Life value
    WriteEEPROM(10, MAXLIVES);        // Blue
    WriteEEPROM(20, MAXLIVES);        // Red

    // Then configure the sound envelope to our pleasing
    WriteOneByteWMask(30, 0xf0, 0xe0);
    WriteOneByteWMask(30, 15, 244);

    // Stop the sound
    ReleaseOscillator(EO_A1);

    // Main loop.
    while(1)
    {
        // Wait for the RTI flag to be set.
        if(RTIFlag)
        {
            RTIFlag = 0;    // First thing we do is clear the flag.

            /* User code goes here. */

            // Get the controller input
            GetContData();

            switch(Status)
            {

              // Waiting for start state, user must hit start to start the game
              case WFSTART:
                if((m_Blue_Life == 0) || (m_Red_Life == 0))
                {
                    WriteEEPROM(10, MAXLIVES);        // Blue
                    WriteEEPROM(20, MAXLIVES);        // Red

                    // Wait for about 2s for the final score.
                    for(index=0;index<60;index++)
                    {
                        while(RTIFlag == 0);    // Wait for RTIFlag to be set.
                        RTIFlag = 0;            // Clear it.
                    }


                    // Write the SPLASH Waiting for Start screen
                    ClearScreen(BLACK);
                    for(index=0;index<20;index++)
                        PlaceCharacter(TitleString[index], 0, 2, GREEN, 15+(15*index), 80);

                    for(index=0;index<18;index++)
                        PlaceCharacter(PSString[index], 0, 1, GREEN, 70+(10*index), 140);

                    // Get the lives remaining
```

```
        m_Blue_Life = ReadEEPROM(10);
        m_Red_Life = ReadEEPROM(20);

        // Start intro music
        MStart = 1;
        MTimeLeft = 0;
        MIndex = 0;
    }


    // Play intro music
    if(MStart == 1)
    {
        if(MTimeLeft == 0)
        {
            // Check Index
            if(MIndex > 12)
                MIndex = 0;
            MTimeLeft = SCMTime[MIndex];
            LoadPlayNote(EO_A1, SCMusic[MIndex]);
            MIndex = MIndex + 1;

        }
        else
        {
            MTimeLeft = MTimeLeft - 1;
        }
    }

    Status = StartGame();
    break;

  // Initialize arena state, user has hit start and the game begins
  case STARTED:
    // Stop the sound
    ReleaseOscillator(EO_A1);
    MStart = 0;

    InitArena();

    // Get the lives remaining
    m_Blue_Life = ReadEEPROM(10);
    m_Red_Life = ReadEEPROM(20);

    for(index=0;index<10;index++) {
        unsigned int BlockX = Rand() % 319;
        unsigned char BlockY = Rand() % 239;

        if(((BlockX > 85)  || (BlockX < 79))  &&
           ((BlockX > 245) || (BlockX < 239)) &&
           ((BlockY > 65)  || (BlockY < 59))  &&
           ((BlockY > 185) || (BlockY < 179)))
        {
            ClearArea(GREEN, BlockX, BlockY, BlockX-1, BlockY-1);
        }
    }

    // Light the LED according the the lives remaining (out of 4)
    for(index=0;index<8;index++)
    {
        if(index <= m_Blue_Life-1)
            SetPWMDuty(index, 0xFF);
        else if(index >= (8 - m_Red_Life))
            SetPWMDuty(index, 0xFF);
        else
            SetPWMDuty(index, 0x00);
    }

    Status = RUNNING;
    break;

  // Running State, the game is in progress
  case RUNNING:

    // Write to screen the Blue Player
    ClearArea(BLUE, m_Blue_PosX, m_Blue_PosY,
                    m_Blue_PosX-1, m_Blue_PosY-1);

    // Write to screen the Red Player
    ClearArea(RED, m_Red_PosX, m_Red_PosY,
                   m_Red_PosX-1, m_Red_PosY-1);

    // Parse the controller command
    ParseController();

    // Move the players and get the new status
    Status = MovePlayers();

    // Check the case
    switch(Status)
    {
      case BOTHDIE:
        // Clear screen and write the words Draw
        ClearScreen(BLACK);
        for(index=0;index<9;index++)
```

```
                        PlaceCharacter(TieString[index], 0, 2, 0x3f, 20+(32*index), 120);

                    // Play lose tone
                    //LoadPlayNote(EO_A1, E3)

                    // Write to EEPROM both new life counts
                    WriteEEPROM(10, m_Blue_Life-1);        // Blue
                    WriteEEPROM(20, m_Red_Life-1);          // Red

                    break;
                  case REDDIED:
                    // Clear screen and write the words Wins in blue
                    ClearScreen(BLACK);
                    for(index=0;index<9;index++)
                        PlaceCharacter(BWString[index], 0, 2, BLUE, 20+(32*index), 120);

                    // Play winning tone
                    //LoadPlayNote(EO_A1, C3)

                    // Write to EEPROM both new life counts
                    WriteEEPROM(10, m_Blue_Life);        // Blue
                    WriteEEPROM(20, m_Red_Life-1);          // Red
                    break;
                  case BLUEDIE:
                    // Clear screen and write the words Wins in red
                    ClearScreen(BLACK);
                    for(index=0;index<8;index++)
                        PlaceCharacter(RWString[index], 0, 2, RED, 36+(32*index), 120);

                    // Play winning tone
                    //LoadPlayNote(EO_A1, C3)

                    // Write to EEPROM new blue life count
                    WriteEEPROM(10, m_Blue_Life-1);        // Blue
                    WriteEEPROM(20, m_Red_Life);          // Red
                    break;
                }  // end switch

                // Current game ends, switch back to waiting for start state
                if(Status != RUNNING) {
                    Status = WFSTART;

                    // Stop the sound
                    ReleaseOscillator(EO_A1);
                }
                break;
            }  // end switch status
        }  // end if RTIFlag
    }  // end while

    return 0;
}

char StartGame()
{
    char Start = WFSTART;

    // Start command
    if((ContP1 & 0x10) == 0)
        Start = STARTED;

    // Start command
    if((ContP2 & 0x10) == 0)
        Start = STARTED;

    return Start;
}

void ParseController()
{
    // Right command
    if((ContP1 & 0x01) == 0)
    {
        if(m_Blue_Dir != ETBD_Left)
            m_Blue_Dir = ETBD_Right;
    }

    // Left command
    if((ContP1 & 0x02) == 0)
    {
        if(m_Blue_Dir != ETBD_Right)
            m_Blue_Dir = ETBD_Left;
    }

    // Down command
    if((ContP1 & 0x04) == 0)
    {
        if(m_Blue_Dir != ETBD_Up)
            m_Blue_Dir = ETBD_Down;
    }

    // Up command
    if((ContP1 & 0x08) == 0)
    {
        if(m_Blue_Dir != ETBD_Down)
```

```
            m_Blue_Dir = ETBD_Up;
     }

    // Right command
    if((ContP2 & 0x01) == 0)
    {
        if(m_Red_Dir != ETBD_Left)
            m_Red_Dir = ETBD_Right;
    }

    // Left command
    if((ContP2 & 0x02) == 0)
    {
        if(m_Red_Dir != ETBD_Right)
            m_Red_Dir = ETBD_Left;
    }

    // Down command
    if((ContP2 & 0x04) == 0)
    {
        if(m_Red_Dir != ETBD_Up)
            m_Red_Dir = ETBD_Down;
    }

    // Up command
    if((ContP2 & 0x08) == 0)
    {
        if(m_Red_Dir != ETBD_Down)
            m_Red_Dir = ETBD_Up;
    }
}

// Initialize the arena and the players
void InitArena()
{

    // Clear all of the playing field on screen
    ClearScreen(BLACK);

    // Draw Boarder
    DrawLine(GREEN, 0, 0, 319, 0);
    DrawLine(GREEN, 0, 0, 0, 239);
    DrawLine(GREEN, 0, 239, 319, 239);
    DrawLine(GREEN, 319, 0, 319, 239);

    // Set up the blue player
    m_Blue_PosX = 80;
    m_Blue_PosY = 60;
    m_Blue_Color = ETBC_Blue;
    m_Blue_Dir = ETBD_Right;

    // Set up the red player
    m_Red_PosX = 240;
    m_Red_PosY = 180;
    m_Red_Color = ETBC_Red;
    m_Red_Dir = ETBD_Left;

    // Play start tone
    LoadPlayNote(EO_A1, C3);
}

// Move the players and determine if they have not run into the wall or
// the other player's trail.
char MovePlayers() {

    char Result = RUNNING;

    LoadPlayNote(EO_A1, F3);

    // Move the blue player
    switch(m_Blue_Dir)
    {
      case ETBD_Up:
        m_Blue_PosY = m_Blue_PosY - 2;
        break;
      case ETBD_Right:
        m_Blue_PosX = m_Blue_PosX + 2;
        break;
      case ETBD_Down:
        m_Blue_PosY = m_Blue_PosY + 2;
        break;
      case ETBD_Left:
        m_Blue_PosX = m_Blue_PosX - 2;
        break;
    }

    // Check if blue player is still alive
    if((m_Blue_PosX > 1) && (m_Blue_PosX < ARENAWIDTH-1) &&
       (m_Blue_PosY > 1) && (m_Blue_PosY < ARENAHEIGHT-1))
    {
        // Check the lower right of the current move
        ReadPixel(m_Blue_PosX, m_Blue_PosY);
        if(ezVGAReturn != BLACK)
        {
            Result = BLUEDIE;
```

```
            // Play wall tone
            LoadPlayNote(EO_A1, D3);

        }
        // Check the upper left of the current move
        ReadPixel(m_Blue_PosX, m_Blue_PosY);
        if(ezVGAReturn != BLACK)
        {
            Result = BLUEDIE;

            // Play wall tone
            LoadPlayNote(EO_A1, D3);

        }
    }
    else
    {
        Result = BLUEDIE;

        // Play crash tone
        LoadPlayNote(EO_A1, E3);
    }

    // Move the red player
    switch(m_Red_Dir)
    {
      case ETBD_Up:
        m_Red_PosY = m_Red_PosY - 2;
        break;
      case ETBD_Right:
        m_Red_PosX = m_Red_PosX + 2;
        break;
      case ETBD_Down:
        m_Red_PosY = m_Red_PosY + 2;
        break;
      case ETBD_Left:
        m_Red_PosX = m_Red_PosX - 2;
        break;
    }

    // Check if red player is still alive
    if((m_Red_PosX > 1) && (m_Red_PosX < ARENAWIDTH-1) &&
       (m_Red_PosY > 1) && (m_Red_PosY < ARENAHEIGHT-1))
    {
        // Check lower right of the current move
        ReadPixel(m_Red_PosX, m_Red_PosY);
        if(ezVGAReturn != BLACK)
        {
            if(Result == BLUEDIE)
                Result = BOTHDIE;
            else
                Result = REDDIED;

            // Play wall tone
            LoadPlayNote(EO_A1, D3);

        }
        // Check the upper left of the current move
        ReadPixel(m_Red_PosX-1, m_Red_PosY-1);
        if(ezVGAReturn != BLACK)
        {
            if(Result == BLUEDIE)
                Result = BOTHDIE;
            else
                Result = REDDIED;

            // Play wall tone
            LoadPlayNote(EO_A1, D3);

        }
    }
    else
    {
        if(Result == BLUEDIE)
            Result = BOTHDIE;
        else
            Result = REDDIED;

        // Play crash tone
        LoadPlayNote(EO_A1, E3);
    }

    // Check case where the two occupies the same space
    if((m_Red_PosX == m_Blue_PosX) && (m_Red_PosY == m_Blue_PosY)) {
        Result = BOTHDIE;
    }

    return Result;
}
```

3. Kill Kevin Specific Code

```
int main()
{
    // ------------------------------------------------------------------
    // Here we have the data needed for the Splash Screen State machine.
    // ------------------------------------------------------------------
    // Splash Screen state machine variable.
    volatile unsigned int SplashState = 0;

    // Splash Screen strings that are displayed.
    char KKString[10] = "KILL KEVIN";
    char PSString[18] = "Push Start to Play";
    char PSelString[19] = "Push Select to Load";
    char P1ScoreString[11] = "P1 Score = ";
    char P2ScoreString[11] = "P2 Score = ";
    char ReadyString[5] = "READY";

    // Index for displaying strings.
    volatile int SplashIndex=0;

    // RTI counter for keeping track of time.
    volatile unsigned int SplashRTI = 0;
    volatile unsigned int VeryEndRTI = 0;

    // ------------------------------------------------------------------
    // Here we have the data needed for the Game Music State machine.
    // ------------------------------------------------------------------
    // Intro music is from 2001. Here are the notes and their duration.
    unsigned char IntroNote[18] = {A4s, F5, A5s, D6, C6s, G5, A5, A5s, C6, D6, D6s, F6, D6, D6s, F6, G6, A6, A6s};
    unsigned int IntroDur[18] =    {60, 60, 120, 15, 75,  15, 15, 60,  30, 15, 15,  90, 8,  8,   60, 40, 60, 80};
    unsigned char WinNote[24] = {D6, C6, D6, C6, A5s, A6s, A6, A6s, A6, G6, A6, F6, A6s, A6, F6, A6s, G6s, F6, A6s,
G6s, F6, D6s, C6, F6};
    unsigned int WinDur[24] =   {23, 15, 8, 23, 23,  23,  15, 8,  23, 23, 23, 15, 8,  23, 15, 8,  23,  15, 8,  23,  30,
8,  8, 60};

    // Index to index the music notes and duration.
    volatile int MusicIndex=0;

    // Counter used to determine if we are done with the music.
    volatile unsigned char NoteCounter = 0;

    // RTI counter for keeping track of time.
    volatile unsigned int SoundRTI = 0;
    volatile unsigned int AnnoyingRTI = 0;

    // Flags for control.
    volatile char GMFlag = 0;
    volatile char WinFlag = 0;
    volatile char AnnoyingFlag = 0;
    volatile char GNFlag = 0;

    // Music state machine variable.
    volatile unsigned int MusicState = 0;

    // Variable for the dynamic duration of the Game Music.
    volatile unsigned char GDur;

    // Temp for Random numbers.
    volatile unsigned int SplashTemp;

    // ------------------------------------------------------------------
    // Here we have the data needed for the Select State machine.
    // ------------------------------------------------------------------
    // Select state machine variable.
    volatile unsigned int SelectState = 0;

    // Strings for the Select SM
    char ResumeString[6] =  "Resume";
    char RestoreString[7] = "Restore";
    char SaveString[4] =    "Save";
    char DoneString[4] =    "Done";
    char RTOString[26] = "Kevin is a Right Turn Only";

    // Index for the Select SM strings.
    volatile unsigned char SelIndex;

    // Variable that determine what we are selecting.
    volatile unsigned char SelVar;

    // Variable for the Select SM RTI counter.
    volatile unsigned char SelRTI;

    // ------------------------------------------------------------------
    // Here we have the data needed for the Player State machines.
    // ------------------------------------------------------------------
    // Strings for Player Wins.
    char P1WinString[9] = "P1 WINS!!";
    char P2WinString[9] = "P2 WINS!!";

    // Indexes for the strings.
```

```
volatile unsigned char P1Index, P2Index;

// Select state machine variable.
volatile unsigned int Player1State = 0;
volatile unsigned int Player2State = 0;

// Temp variable.
volatile unsigned int P1Temp,P2Temp;

// RTI counter for keeping track of time.
volatile unsigned int P1RTI = 0;
volatile unsigned int P2RTI = 0;

// Color Array.
unsigned char ColorArray[6] = {0x1f, 0x0c, 0x07, 0x22, 0x3d, 0x1b};
volatile unsigned char P1CIndex, P2CIndex;

// ----------------------------------------------------------------
// Here we have variables used by multiple state machines.
// ----------------------------------------------------------------
// Flag to indicate that we have in fact started.
volatile unsigned char Started=0;

// Flag to indicate that we don't need get random number as we have loaded.
volatile unsigned char Loaded=0;

// Random Player indexes that indicate where we are in the Random Array.
volatile unsigned char P1RandIndex=0, P2RandIndex=0;

// Random Array that stores the random numbers.
unsigned int RandArray[NUMOFRAND];

// First call the Init function to intialize all the IO.
Init();

// Then configure the sound envelope to our pleasing
WriteOneByteWMask(30, 0xf0, 0xe0);
WriteOneByteWMask(30, 15, 244);

// Main loop.
while(1)
{
    // Wait for the RTI flag to be set.
    if(RTIFlag)
    {
        RTIFlag = 0;    // First thing we do is clear the flag.

        // We will always get the controller data so do that first.
        GetContData();

        // This is the Splash screen state machine. It will the be the
        // first state machine that will run.
        switch(SplashState)
        {
            case SHalt:     SplashState = SHalt;
                            break;

            // At the very end we wait for the user to push a button. When
            // they do we start the whole game over.
            // First wait for the user to stop pushing a button.
            case VeryEnd:   Loaded=0;Started=0;
                            if(ContP1 == 0xff && ContP2 == 0xff)
                                SplashState = VeryEnd+1;

                            VeryEndRTI = 0;
                            break;

            // Now wait two seconds.
            case VeryEnd+1: if(VeryEndRTI == 60)
                                SplashState = VeryEnd+2;

                            VeryEndRTI++;

                            break;

            // Then wait until we push something.
            case VeryEnd+2: if(ContP1 != 0xff || ContP2 != 0xff)
                            {
                                MusicState = MSHalt;      // Stop the Music SM.
                                SplashState = SState;     // Start all over.
                                SplashState = VeryEnd+3;
                            }
                            break;

            // Then wait until we are done pushing everything.
            case VeryEnd+3: if(ContP1 == 0xff && ContP2 == 0xff)
                                SplashState = SState;
                            break;

            // First we display the strings on the screen.
            case SState:    ClearScreen(0x00);    // Clear the screen.

                            // Draw the Splash Screen strings.
                            for(SplashIndex=0;SplashIndex<10;SplashIndex++)
                                PlaceCharacter(KKString[SplashIndex], 0, 3, 0x03, 50+(20*SplashIndex), 80);
```

```
                for(SplashIndex=0;SplashIndex<18;SplashIndex++)
                    PlaceCharacter(PSString[SplashIndex], 0, 1, 0x3f, 60+(10*SplashIndex), 120);
                for(SplashIndex=0;SplashIndex<19;SplashIndex++)
                    PlaceCharacter(PSelString[SplashIndex], 0, 1, 0x3f, 56+(10*SplashIndex), 140);

                if(Loaded == 0)     // Check to see if we have loaded a game.
                {                   // If we have not then
                    P1RandIndex=0;  // clear the Rand number indexes for both players.
                    P2RandIndex=0;
                }

                MusicState = MIState;         // Start the Music Intro.
                SplashState = SState+1;       // Goto next state.
                break;

// Wait for user input.
case SState+1:  if((ContP1 & START) == 0 || (ContP2 & START) == 0)
                    SplashState = SState + 3;
                if((ContP1 & SEL) == 0 || (ContP2 & SEL) == 0)
                    SplashState = SState + 2;
                break;

// If the user pushes Select then they want to load a game.
case SState+2:  MusicState = MSHalt;     // Halt the music.
                SplashState = SHalt;     // Halt us.
                SelectState = SelStart;  // Start the Select State Machine.
                break;

// If the user pushes start then work towards playing the game.
case SState+3:  MusicState = MSHalt;     // Halt the music.

                Started = 1;             // Indicate that we have started.
                SplashState = SState+4;  // Goto the next state.

                // If we have loaded up a previous game then
                // we don't need to get any more random numbers.
                if(Loaded)
                    break;

                Seed();                  // Seed the Random number generator.

                // Call the Rand() function 16 times to Roll
                // all the way through the shift register one
                // time.
                for(SplashIndex=0;SplashIndex<16;SplashIndex++)
                    Rand();

                // Fill the RandArray with random numbers. Be sure that
                // the numbers are between 0 and 5.
                for(SplashIndex=0;SplashIndex<NUMOFRAND;SplashIndex++)
                {
                    do
                    {
                        SplashTemp = Rand() & 0x07;
                    }
                    while(SplashTemp > 5);

                    RandArray[SplashIndex] = SplashTemp;
                }

                break;

case SState+4:  ClearScreen(0x00);    // Clear the screen.

                // Draw the outline.
                DrawLine(0x2A, 0, 0, 319, 0);
                DrawLine(0x2A, 0, 0, 0, 239);
                DrawLine(0x2A, 0, 239, 319, 239);
                DrawLine(0x2A, 319, 0, 319, 239);
                DrawLine(0x2A, 160, 0, 160, 239);

                // Draw Kill Kevin.
                for(SplashIndex=0;SplashIndex<10;SplashIndex++)
                    PlaceCharacter(KKString[SplashIndex], 0, 1, 0x03, 112+(10*SplashIndex), 5);

                // Draw P1 and P2 Score.
                for(SplashIndex=0;SplashIndex<11;SplashIndex++)
                    PlaceCharacter(P1ScoreString[SplashIndex], 0, 1, 0x3f, 10+(10*SplashIndex), 25);
                for(SplashIndex=0;SplashIndex<11;SplashIndex++)
                    PlaceCharacter(P2ScoreString[SplashIndex], 0, 1, 0x3f, 170+(10*SplashIndex), 25);

                // Draw the Crosses.
                ClearArea(0x3f, 78, 70, 82, 170);
                ClearArea(0x3f, 30, 118, 130, 122);
                ClearArea(0x3f, 238, 70, 242, 170);
                ClearArea(0x3f, 190, 118, 290, 122);

                // Draw Ready.
                for(SplashIndex=0;SplashIndex<5;SplashIndex++)
                    PlaceCharacter(ReadyString[SplashIndex], 0, 3, 0x2E, 109+(20*SplashIndex), 140);

                // Place the countdown. Start with 3.
                PlaceCharacter('3', 0, 3, 0x27, 148, 190);

                // Clear the RTI counter.
```

```
                    SplashRTI = 0;

                    SplashState = SState+5;   // Goto the next state.
                    break;

    // Wait 3 seconds before actually beginning. Countdown the time
    // in seconds. When ready clear Ready and number and halt.
    case SState+5:  if(SplashRTI == 30)
                        PlaceCharacter('2', 0, 3, 0x1B, 148, 190);
                    if(SplashRTI == 60)
                        PlaceCharacter('1', 0, 3, 0x1F, 148, 190);
                    if(SplashRTI == 90)
                    {
                        ClearArea(0x00, 109, 140, 220, 220); // Clear Ready and number.
                        DrawLine(0x2A, 160, 0, 160, 239);    // Redraw our middle line.
                        Player1State = P1State;   // Start P1 SM.
                        Player2State = P2State;   // Start P2 SM.
                        MusicState = MGState;     // Start the Game Music.
                        SplashState = SHalt;      // Halt ourselves.

                        P1CIndex = RandArray[0] & 0x3; // Determine starting Color index.
                        P2CIndex = P1CIndex;
                    }

                    // Increment the RTI counter no matter what.
                    SplashRTI++;

                    break;

}

// This is the Player1 state machine.
switch(Player1State)
{
    // Begin in the halt state. Stay halted until the Splash SM
    // takes us out.
    case Halt:        Player1State = Halt;
                      break;

    // First thing we'll do is update the PWM. The farther we get
    // the more the LEDs will light up.
    case P1State:     // Right most Green LED.
                      if(P1RandIndex < 3 && P1RandIndex > 0)
                          SetPWMDuty(3, 0x40);
                      else if(P1RandIndex < 6 && P1RandIndex > 0)
                          SetPWMDuty(3, 0x80);
                      else if(P1RandIndex < 9 && P1RandIndex > 0)
                          SetPWMDuty(3, 0xc0);
                      else if(P1RandIndex < 12 && P1RandIndex > 0)
                          SetPWMDuty(3, 0xff);
                      else if(P1RandIndex == 0)
                          SetPWMDuty(3, 0x00);

                      // Next Green LED.
                      if(P1RandIndex < 15 && P1RandIndex >= 12)
                          SetPWMDuty(2, 0x40);
                      else if(P1RandIndex < 18 && P1RandIndex >= 12)
                          SetPWMDuty(2, 0x80);
                      else if(P1RandIndex < 21 && P1RandIndex >= 12)
                          SetPWMDuty(2, 0xc0);
                      else if(P1RandIndex < 24 && P1RandIndex >= 12)
                          SetPWMDuty(2, 0xff);
                      else if(P1RandIndex < 12)
                          SetPWMDuty(2, 0x00);

                      // Red LED.
                      if(P1RandIndex < 27 && P1RandIndex >= 24)
                          SetPWMDuty(1, 0x40);
                      else if(P1RandIndex < 30 && P1RandIndex >= 24)
                          SetPWMDuty(1, 0x80);
                      else if(P1RandIndex < 33 && P1RandIndex >= 24)
                          SetPWMDuty(1, 0xc0);
                      else if(P1RandIndex < 36 && P1RandIndex >= 24)
                          SetPWMDuty(1, 0xff);
                      else if(P1RandIndex < 24)
                          SetPWMDuty(1, 0x00);

                      // Blue LED.
                      if(P1RandIndex < 39 && P1RandIndex >= 36)
                          SetPWMDuty(0, 0x40);
                      else if(P1RandIndex < 42 && P1RandIndex >= 36)
                          SetPWMDuty(0, 0x80);
                      else if(P1RandIndex < 45 && P1RandIndex >= 36)
                          SetPWMDuty(0, 0xc0);
                      else if(P1RandIndex < 36)
                          SetPWMDuty(0, 0x00);
                      else
                          SetPWMDuty(0, 0xff);

                      Player1State = P1State+1;
                      break;

    // Now display the tens digit of the score.
    case P1State+1:   if(P1RandIndex < 10)
                          PlaceCharacter('0', 0, 1, 0x3F, 120, 25);
```

```
                    else if(P1RandIndex < 20)
                        PlaceCharacter('1', 0, 1, 0x3F, 120, 25);
                    else if(P1RandIndex < 30)
                        PlaceCharacter('2', 0, 1, 0x3F, 120, 25);
                    else if(P1RandIndex < 40)
                        PlaceCharacter('3', 0, 1, 0x3F, 120, 25);
                    else if(P1RandIndex < 50)
                        PlaceCharacter('4', 0, 1, 0x3F, 120, 25);
                    else
                        PlaceCharacter('5', 0, 1, 0x3F, 120, 25);

                    Player1State = P1State+2;
                    break;

  // Now display the ones digit of the score.
  case P1State+2:   if(P1RandIndex < 10)
                        P1Temp = P1RandIndex+0x30;
                    else if(P1RandIndex < 20)
                        P1Temp = (P1RandIndex-10) + 0x30;
                    else if(P1RandIndex < 30)
                        P1Temp = (P1RandIndex-20) + 0x30;
                    else if(P1RandIndex < 40)
                        P1Temp = (P1RandIndex-30) + 0x30;
                    else if(P1RandIndex < 50)
                        P1Temp = (P1RandIndex-40) + 0x30;
                    else
                        P1Temp = 0x30;

                    PlaceCharacter(P1Temp, 0, 1, 0x3F, 130, 25);
                    Player1State = P1State+3;
                    break;

  // Clear upper and lower boxes.
  case P1State+3:   ClearArea(0x00, 78, 64, 82, 68);
                    ClearArea(0x00, 78, 172, 82, 176);

                    Player1State = P1State+4;
                    break;

  // Clear left and right boxes.
  case P1State+4:   ClearArea(0x00, 24, 118, 28, 122);
                    ClearArea(0x00, 132, 118, 136, 122);

                    P1RTI = 0;
                    Player1State = P1State+5;
                    break;

  // Wait for player to not push any buttons.
  case P1State+5:   if(ContP1 == 0xff)
                        Player1State = P1State+6;

                    break;

  // Determine color and location of box(es).
  case P1State+6:   P1CIndex++;        // Increment color index to next color.
                    if(P1CIndex == 6)  // If we raech six then roll over to zero.
                        P1CIndex = 0;

                    // Based on the random number determine what
                    // boxes we should draw in.
                    if(RandArray[P1RandIndex] == 0)
                        ClearArea(ColorArray[P1CIndex], 78, 64, 82, 68);
                    else if(RandArray[P1RandIndex] == 1)
                        ClearArea(ColorArray[P1CIndex], 78, 172, 82, 176);
                    else if(RandArray[P1RandIndex] == 2)
                    {
                        ClearArea(ColorArray[P1CIndex], 78, 64, 82, 68);
                        ClearArea(ColorArray[P1CIndex], 78, 172, 82, 176);
                    }
                    else if(RandArray[P1RandIndex] == 3)
                        ClearArea(ColorArray[P1CIndex], 24, 118, 28, 122);
                    else if(RandArray[P1RandIndex] == 4)
                        ClearArea(ColorArray[P1CIndex], 132, 118, 136, 122);
                    else if(RandArray[P1RandIndex] == 5)
                    {
                        ClearArea(ColorArray[P1CIndex], 24, 118, 28, 122);
                        ClearArea(ColorArray[P1CIndex], 132, 118, 136, 122);
                    }


                    Player1State = P1State+7;
                    break;

  // Wait for user input.
  case P1State+7:   if(ContP1 == 0xff)  // If user have input nothing.
                    {
                        Player1State = P1State+7;
                        break;
                    }
                    else if((ContP1 & START) == 0 || (ContP1 & SEL) == 0) // Check for start or select.
                    {
                        Player2State = Halt;     // If we get a start or sel then
                        Player1State = Halt;     // Halt both player SMs, the Music SM
                        MusicState = MSHalt;     // and start the Select SM.
                        SelectState = SelStart;
```

```
                        break;
                    }

                    // Else check to see if what the user input was correct.
                    else if((ContP1 & UP) == 0 && RandArray[P1RandIndex] == 0)
                        P1RandIndex++;
                    else if((ContP1 & DOWN) == 0 && RandArray[P1RandIndex] == 1)
                        P1RandIndex++;
                    else if((ContP1 & B) == 0 && RandArray[P1RandIndex] == 2)
                        P1RandIndex++;
                    else if((ContP1 & LEFT) == 0 && RandArray[P1RandIndex] == 3)
                        P1RandIndex++;
                    else if((ContP1 & RIGHT) == 0 && RandArray[P1RandIndex] == 4)
                        P1RandIndex++;
                    else if((ContP1 & A) == 0 && RandArray[P1RandIndex] == 5)
                        P1RandIndex++;

                    // Else the input was incorrect so pay the penalty.
                    else
                    {
                        // Must go back a certain amount.
                        if(P1RandIndex < BACKAMOUNT)
                            P1RandIndex=0;
                        else
                            P1RandIndex -= BACKAMOUNT;

                        // Play an annoying sound to indicate mistake.
                        AnnoyingFlag = 1;
                    }

                    // See if we are done.
                    if(P1RandIndex == NUMOFRAND)
                    {
                        // Display the 50
                        PlaceCharacter('5', 0, 1, 0x3F, 120, 25);
                        PlaceCharacter('0', 0, 1, 0x3F, 130, 25);

                        // Display P1 WINS!!
                        for(P1Index=0;P1Index<9;P1Index++)
                            PlaceCharacter(P1WinString[P1Index], 0, 1, 0x2E, 121+(10*P1Index), 140);

                        // If we are then set the win flag and Halt.
                        WinFlag = 1;
                        Player2State = Halt;
                        Player1State = Halt;
                        SplashState = VeryEnd;
                    }
                    else
                        Player1State = P1State;

                    break;
}

// This is the Player2 state machine.
switch(Player2State)
{
    // Begin in the halt state. Stay halted until the Splash SM
    // takes us out.
    case Halt:      Player2State = Halt;
                    break;

    // First thing we'll do is update the PWM. The farther we get
    // the more the LEDs will light up.
    case P2State:   // Right most Green LED.
                    if(P2RandIndex < 3 && P2RandIndex > 0)
                        SetPWMDuty(7, 0x40);
                    else if(P2RandIndex < 6 && P2RandIndex > 0)
                        SetPWMDuty(7, 0x80);
                    else if(P2RandIndex < 9 && P2RandIndex > 0)
                        SetPWMDuty(7, 0xc0);
                    else if(P2RandIndex < 12 && P2RandIndex > 0)
                        SetPWMDuty(7, 0xff);
                    else if(P2RandIndex == 0)
                        SetPWMDuty(7, 0x00);

                    // Next Green LED.
                    if(P2RandIndex < 15 && P2RandIndex >= 12)
                        SetPWMDuty(6, 0x40);
                    else if(P2RandIndex < 18 && P2RandIndex >= 12)
                        SetPWMDuty(6, 0x80);
                    else if(P2RandIndex < 21 && P2RandIndex >= 12)
                        SetPWMDuty(6, 0xc0);
                    else if(P2RandIndex < 24 && P2RandIndex >= 12)
                        SetPWMDuty(6, 0xff);
                    else if(P2RandIndex < 12)
                        SetPWMDuty(6, 0x00);

                    // Red LED.
                    if(P2RandIndex < 27 && P2RandIndex >= 24)
                        SetPWMDuty(5, 0x40);
                    else if(P2RandIndex < 30 && P2RandIndex >= 24)
                        SetPWMDuty(5, 0x80);
                    else if(P2RandIndex < 33 && P2RandIndex >= 24)
                        SetPWMDuty(5, 0xc0);
                    else if(P2RandIndex < 36 && P2RandIndex >= 24)
```

```
                    SetPWMDuty(5, 0xff);
                else if(P2RandIndex < 24)
                    SetPWMDuty(5, 0x00);

                // Blue LED.
                if(P2RandIndex < 39 && P2RandIndex >= 36)
                    SetPWMDuty(4, 0x40);
                else if(P2RandIndex < 42 && P2RandIndex >= 36)
                    SetPWMDuty(4, 0x80);
                else if(P2RandIndex < 45 && P2RandIndex >= 36)
                    SetPWMDuty(4, 0xc0);
                else if(P2RandIndex < 36)
                    SetPWMDuty(4, 0x00);
                else
                    SetPWMDuty(4, 0xff);

                Player2State = P2State+1;
                break;

// Now display the tens digit of the score.
case P2State+1: if(P2RandIndex < 10)
                    PlaceCharacter('0', 0, 1, 0x3F, 280, 25);
                else if(P2RandIndex < 20)
                    PlaceCharacter('1', 0, 1, 0x3F, 280, 25);
                else if(P2RandIndex < 30)
                    PlaceCharacter('2', 0, 1, 0x3F, 280, 25);
                else if(P2RandIndex < 40)
                    PlaceCharacter('3', 0, 1, 0x3F, 280, 25);
                else if(P2RandIndex < 50)
                    PlaceCharacter('4', 0, 1, 0x3F, 280, 25);
                else
                    PlaceCharacter('5', 0, 1, 0x3F, 280, 25);

                Player2State = P2State+2;
                break;

// Now display the ones digit of the score.
case P2State+2: if(P2RandIndex < 10)
                    P2Temp = P2RandIndex+0x30;
                else if(P2RandIndex < 20)
                    P2Temp = (P2RandIndex-10) + 0x30;
                else if(P2RandIndex < 30)
                    P2Temp = (P2RandIndex-20) + 0x30;
                else if(P2RandIndex < 40)
                    P2Temp = (P2RandIndex-30) + 0x30;
                else if(P2RandIndex < 50)
                    P2Temp = (P2RandIndex-40) + 0x30;
                else
                    P2Temp = 0x30;

                PlaceCharacter(P2Temp, 0, 1, 0x3F, 290, 25);
                Player2State = P2State+3;
                break;

// Clear upper and lower boxes.
case P2State+3: ClearArea(0x00, 238, 64, 242, 68);
                ClearArea(0x00, 238, 172, 242, 176);

                Player2State = P2State+4;
                break;

// Clear left and right boxes.
case P2State+4: ClearArea(0x00, 184, 118, 188, 122);
                ClearArea(0x00, 292, 118, 296, 122);

                P2RTI = 0;
                Player2State = P2State+5;
                break;

// Wait for player to not push any buttons.
case P2State+5: if(ContP2 == 0xff)
                    Player2State = P2State+6;

                break;

// Determine color and location of box(es).
case P2State+6: P2CIndex++;        // Increment color index to next color.
                if(P2CIndex == 6)  // If we raech six then roll over to zero.
                    P2CIndex = 0;

                // Based on the random number determine what
                // boxes we should draw in.
                if(RandArray[P2RandIndex] == 0)
                    ClearArea(ColorArray[P2CIndex], 238, 64, 242, 68);
                else if(RandArray[P2RandIndex] == 1)
                    ClearArea(ColorArray[P2CIndex], 238, 172, 242, 176);
                else if(RandArray[P2RandIndex] == 2)
                {
                    ClearArea(ColorArray[P2CIndex], 238, 64, 242, 68);
                    ClearArea(ColorArray[P2CIndex], 238, 172, 242, 176);
                }
                else if(RandArray[P2RandIndex] == 3)
                    ClearArea(ColorArray[P2CIndex], 184, 118, 188, 122);
                else if(RandArray[P2RandIndex] == 4)
                    ClearArea(ColorArray[P2CIndex], 292, 118, 296, 122);
```

```
                        else if(RandArray[P2RandIndex] == 5)
                        {
                            ClearArea(ColorArray[P2CIndex], 184, 118, 188, 122);
                            ClearArea(ColorArray[P2CIndex], 292, 118, 296, 122);
                        }


                        Player2State = P2State+7;
                        break;

    // Wait for user input.
    case P2State+7:     if(ContP2 == 0xff)  // If user have input nothing.
                        {
                            Player2State = P2State+7;
                            break;
                        }
                        else if((ContP2 & START) == 0 || (ContP2 & SEL) == 0) // Check for start or select.
                        {
                            Player2State = Halt;     // If we get a start or sel then
                            Player1State = Halt;     // Halt both player SMs, the Music SM
                            MusicState = MSHalt;     // and start the Select SM.
                            SelectState = SelStart;
                            break;
                        }

                        // Else check to see if what the user input was correct.
                        else if((ContP2 & UP) == 0 && RandArray[P2RandIndex] == 0)
                            P2RandIndex++;
                        else if((ContP2 & DOWN) == 0 && RandArray[P2RandIndex] == 1)
                            P2RandIndex++;
                        else if((ContP2 & B) == 0 && RandArray[P2RandIndex] == 2)
                            P2RandIndex++;
                        else if((ContP2 & LEFT) == 0 && RandArray[P2RandIndex] == 3)
                            P2RandIndex++;
                        else if((ContP2 & RIGHT) == 0 && RandArray[P2RandIndex] == 4)
                            P2RandIndex++;
                        else if((ContP2 & A) == 0 && RandArray[P2RandIndex] == 5)
                            P2RandIndex++;

                        // Else the input was incorrect so pay the penalty.
                        else
                        {
                            // Must go back a certain amount.
                            if(P2RandIndex < BACKAMOUNT)
                                P2RandIndex=0;
                            else
                                P2RandIndex -= BACKAMOUNT;

                            // Play an annoying sound to indicate mistake.
                            AnnoyingFlag = 1;
                        }

                        // See if we are done.
                        if(P2RandIndex == NUMOFRAND)
                        {
                            // Display the 50
                            PlaceCharacter('5', 0, 1, 0x3F, 280, 25);
                            PlaceCharacter('0', 0, 1, 0x3F, 290, 25);

                            // Display P2 WINS!!
                            for(P2Index=0;P2Index<9;P2Index++)
                                PlaceCharacter(P2WinString[P2Index], 0, 1, 0x2E, 121+(10*P2Index), 140);

                            // If we are then set the win flag and Halt.
                            WinFlag = 1;
                            Player2State = Halt;
                            Player1State = Halt;
                            SplashState = VeryEnd;
                        }
                        else
                            Player2State = P2State;

                        break;
}


// This is the Music State Machine. It will play the intro tune,
// the winning tune, the gameplay tune, and the annoying tune.
switch(MusicState)
{
    // This is the state the SM starts up. Does nothing.
    case Halt:          MusicState = Halt;
                        break;

    // This is the Halt state that any other state machine places
    // this SM in. It will release the osc and then wait for the
    // MGFlag (Music Go Flag) to be set.
    case MSHalt:        ReleaseOscillator(EO_A1); // Turns off the sound.
                        MusicIndex = 0;           // Clear the index.
                        NoteCounter = 0;          // Clear the note counter.
                        MusicState = MSHalt+1;    // Goto next state.
                        break;

    case MSHalt+1:      if(GMFlag)                // If the Flag gets set
                            MusicState = MGState; // then play the Go Music.
```

```
                  break;

// This is the part where we play the intro music. It is the
// 2001 Intro tune.
case MIState:     if(GMFlag) // Check to see if we should play
                  {          // Go Music.
                      ReleaseOscillator(EO_A1);  // Turn off sound.
                      MusicState = MGState;      // Play the Go Music.
                      break;
                  }
                  LoadPlayNote(EO_A1,IntroNote[MusicIndex]);  // Play the next note.
                  SoundRTI = 0;                               // Clear the RTI counter.
                  MusicState = MIState+1;                     // Goto next state.
                  break;

case MIState+1:   if(SoundRTI == IntroDur[MusicIndex]) // Determine how long to play note.
                  {
                      // If we are done playing the note then
                      // determine if we should play another.
                      NoteCounter++;
                      if(NoteCounter == IntroNotes)
                      {
                          // If we are done playing notes then Halt.
                          MusicState = MSHalt;  // Goto Halt state.
                          MusicIndex = 0;       // Clear the music index.
                      }
                      else
                      {
                          // If we are not done then index then next note.
                          MusicIndex++;         // Increment index to play next note.
                          MusicState = MIState; // Goto the state that will play next note.
                      }
                  }

                  // Else we are not done playing the note so
                  // increment the RTI counter.
                  else
                      SoundRTI++;

                  break;


// This is the portion of the state machine that will play
// the winning tune. It will play "We are the Champions."
case MWState:     WinFlag = 0;                              // Clear the Flag that got us here.
                  LoadPlayNote(EO_A1,WinNote[MusicIndex]); // Play the next note.
                  SoundRTI = 0;                             // Clear RTI counter.
                  MusicState = MWState+1;                   // Goto next state.
                  break;

case MWState+1:   if(SoundRTI == WinDur[MusicIndex])  // Determine how long we play note.
                  {
                      // If we are done playing note determine if we are done with tune.
                      NoteCounter++;
                      if(NoteCounter == WinNotes)  // Determines if we are done with tune.
                      {
                          MusicState = MSHalt;     // If we are then Halt.
                          MusicIndex = 0;          // Reset the index that says what note to play.
                      }
                      else   // Else we are not done with tune.
                      {
                          MusicIndex++;            // Increment index to play the next tune.
                          MusicState = MWState;    // Goto the state that plays next tune.
                      }
                  }

                  // Else we are not done playing note.
                  else
                      SoundRTI++;  // Increment the RTI counter.

                  break;

// This is the state that will play the annoying sound when you
// make a mistake.
case MAState:     WriteOneByteWMask(16, 7, 4);  // Change to a square wave.
                  MusicState = MAState+1;       // Goto next state.
                  break;

case MAState+1:   LoadPlayNote(EO_A1,C2);       // Play low note.
                  AnnoyingRTI = 0;              // Clear Annoying RTI counter.
                  MusicState = MAState+2;       // Goto next state.
                  break;

case MAState+2:   if(AnnoyingRTI == 8)          // Determine if we are done playing note.
                  {
                      WriteOneByteWMask(16, 7, 0); // If we are then change to sine wave.
                      MusicState = MGState;        // Goto Game Music state.
                      AnnoyingFlag = 0;            // Clear the flag that got us here.
                  }

                  // Else we are not done playing music so increment the RTI counter.
                  else
                      AnnoyingRTI++;

                  break;
```

```
    // The is the Music Go state where we play the music that is
    // heard during actual game play.
    case MGState:     GMFlag = 0;                 // Clear the flag that got us here.
                      if(GNFlag)                  // Check to see which note we play.
                      {
                          LoadPlayNote(EO_A1,C5); // Play C5 note.
                          GNFlag = 0;             // Change flag to play other note next time.
                      }
                      else
                      {
                          LoadPlayNote(EO_A1,C5s);// Play C5 sharp note.
                          GNFlag = 1;             // Change flag to play other note next time.
                      }
                      SoundRTI = 0;               // Clear the RTI counter.
                      MusicState = MGState+1;     // Goto next state.
                      break;

    case MGState+1:   if(P1RandIndex > P2RandIndex) // Determine our duration based on the
                          GDur = P1RandIndex >> 1;  // PRandIndexes. As the player get closer
                      else                          // to the end the duration gets shorter.
                          GDur = P2RandIndex >> 1;

                      GDur = 27 - GDur;
                      MusicState = MGState+2;
                      break;

    case MGState+2:   if(SoundRTI == GDur)          // Determine if we are done playing note.
                          MusicState = MGState;     // If so then goto state that play the next note.
                      else                          // Else we are not done play the note
                          SoundRTI++;               // so just increment the RTI counter.

                      if(AnnoyingFlag)              // Check to see if we should play the annoying note.
                      {
                          MusicState = MAState;
                      }
                      else if(WinFlag)
                      {
                          MusicState = MWState;     // Check if we should play the winning tune.
                      }

                      break;

}

// When we hit start or select then we get a menu to save and restore games.
// This is the state machine which controls that.
switch(SelectState)
{
    // Halt state which just sits here and does nothing.
    case Halt:        SelectState = Halt;
                      break;

    // Start by displaying the Resume, Restore, and Save strings.
    case SelStart:    ClearScreen(0x00);
                      for(SelIndex=0;SelIndex<6;SelIndex++)
                          PlaceCharacter(ResumeString[SelIndex], 0, 2, 0x3f, 90+(20*SelIndex), 80);
                      for(SelIndex=0;SelIndex<7;SelIndex++)
                          PlaceCharacter(RestoreString[SelIndex], 0, 2, 0x3f, 90+(20*SelIndex), 110);
                      for(SelIndex=0;SelIndex<4;SelIndex++)
                          PlaceCharacter(SaveString[SelIndex], 0, 2, 0x3f, 90+(20*SelIndex), 140);

                      SelVar = 0;                // Clear the var that indicates out selection.

                      SelectState = SelStart+1;  // Goto next state.

                      /*
                      // This is just for fun. Best to comment out later.
                      for(SelIndex=0;SelIndex<26;SelIndex++)
                          PlaceCharacter(RTOString[SelIndex], 0, 1, 0x37, 30+(10*SelIndex), 20);

                      // Draw ECE625 Spike Lab Logo.
                      WritePixel(0x06, FUNX, FUNY);
                      DrawLine(0x1b, FUNX, FUNY-4, FUNX-2, FUNY-6);
                      DrawLine(0x1b, FUNX-2, FUNY-6, FUNX-10, FUNY-6);
                      DrawLine(0x1b, FUNX-10, FUNY-6, FUNX-14, FUNY-4);
                      DrawLine(0x1b, FUNX-14, FUNY-4, FUNX-14, FUNY+2);
                      DrawLine(0x1b, FUNX-14, FUNY+2, FUNX-10, FUNY+6);
                      DrawLine(0x1b, FUNX-10, FUNY+6, FUNX-14, FUNY+26);
                      DrawLine(0x1b, FUNX-8, FUNY+26, FUNX, FUNY+6);
                      DrawLine(0x1b, FUNX, FUNY+6, FUNX+8, FUNY+26);
                      DrawLine(0x1b, FUNX+14, FUNY+26, FUNX+10, FUNY+6);
                      DrawLine(0x1b, FUNX+10, FUNY+6, FUNX+14, FUNY+2);
                      DrawLine(0x1b, FUNX+14, FUNY+2, FUNX+14, FUNY-4);
                      DrawLine(0x1b, FUNX+14, FUNY-4, FUNX+10, FUNY-6);
                      DrawLine(0x1b, FUNX+10, FUNY-6, FUNX+2, FUNY-6);
                      DrawLine(0x1b, FUNX+2, FUNY-6, FUNX, FUNY-4);

                      WritePixel(0x31, FUNX-4, FUNY+16);
                      WritePixel(0x31, FUNX-4, FUNY+17);
                      DrawLine(0x31, FUNX-3, FUNY+18, FUNX+3, FUNY+18);
                      DrawLine(0x31, FUNX+3, FUNY+18, FUNX+4, FUNY+16);
                      DrawLine(0x31, FUNX, FUNY+18, FUNX, FUNY+17);
                      */
```

```
                    break;

// Based on the value of SelVar place the cursor.
case SelStart+1:  PlaceCharacter(' ', 0 , 1, 0x3f, 80, 83);  // Clear all possible
                  PlaceCharacter(' ', 0 , 1, 0x3f, 80, 113); // cursor locations.
                  PlaceCharacter(' ', 0 , 1, 0x3f, 80, 143);

                  // Draw in the cursor in the correct place.
                  if(SelVar == 0)
                      PlaceCharacter('o', 0 , 1, 0x3f, 80, 83);
                  else if(SelVar == 1)
                      PlaceCharacter('o', 0 , 1, 0x3f, 80, 113);
                  else if(SelVar == 2)
                      PlaceCharacter('o', 0 , 1, 0x3f, 80, 143);

                  SelectState = SelStart+2;
                  break;

// Wait for user input.
case SelStart+2:  if((ContP1 & DOWN) == 0 || (ContP2 & DOWN) == 0)
                  {
                      // If the user pushed down then increment
                      // the SelVar value if we can.
                      if(SelVar < 2)
                          SelVar++;

                      SelectState = SelStart+3;
                  }
                  else if((ContP1 & UP) == 0 || (ContP2 & UP) == 0)
                  {
                      // If the user pushed up then decrement
                      // the SelVar value if we can.
                      if(SelVar > 0)
                          SelVar--;

                      SelectState = SelStart+3;
                  }

                  // If the user pushes start or A then execute
                  // whatever the cursor is pointing to.
                  else if((ContP1 & START) == 0 || (ContP2 & START) == 0)
                  {
                      SelectState = SelStart+4;
                  }
                  else if((ContP1 & A) == 0 || (ContP2 & A) == 0)
                  {
                      SelectState = SelStart+4;
                  }

                  break;

// Next two state just wait for the user to stop pushing a
// button before moving on.
case SelStart+3:  if(ContP1 == 0xff && ContP2 == 0xff)
                  {
                      SelectState = SelStart+1;
                  }
                  break;

case SelStart+4:  if(ContP1 == 0xff && ContP2 == 0xff)
                  {
                      SelectState = SelStart+5;
                  }
                  break;

// Take action based on the value of SelVar.
case SelStart+5:  if(SelVar == 0)
                  {
                      // If we sould resume then halt us. Depending
                      // on whether we have already started or not
                      // depends on where we start the Splash SM at.
                      SelectState = Halt;
                      if(Started)
                          SplashState = SState+4;
                      else
                          SplashState = SState;
                      break;
                  }

                  // Restore a previous game.
                  else if(SelVar == 1)
                  {
                      // Restore the Player Indexes and the RandArray.
                      P1RandIndex = ReadEEPROM(0x00);
                      P2RandIndex = ReadEEPROM(0x01);
                      for(SelIndex=0;SelIndex<NUMOFRAND;SelIndex++)
                          RandArray[SelIndex] = ReadEEPROM(2+SelIndex);

                      Loaded = 1;
                  }

                  // Save a game.
                  else if(SelVar == 2)
                  {
                      // Save the Player Indexes and the RandArray.
```

```
                                WriteEEPROM(0x00, P1RandIndex);
                                WriteEEPROM(0x01, P2RandIndex);
                                for(SelIndex=0;SelIndex<NUMOFRAND;SelIndex++)
                                    WriteEEPROM(2+SelIndex, RandArray[SelIndex]);
                            }

                            SelRTI = 0;  // Clear the RTI counter.
                            SelectState = SelStart+6;
                            break;

            // Display a done string to let the user know that the action was completed.
            case SelStart+6:  for(SelIndex=0;SelIndex<4;SelIndex++)
                                PlaceCharacter(DoneString[SelIndex], 0, 2, 0x3f, 90+(20*SelIndex), 200);

                            SelectState = SelStart+7;
                            break;

            // Display the string for 1 second.
            case SelStart+7:  if(SelRTI == 30)
                            {
                                ClearArea(0x00, 90, 200, 200, 220); // Clear the done.
                                SelectState = SelStart+1;
                            }

                            SelRTI++;
                            break;

            }
        }
    }

    return 0;
}
```