University of Oklahoma

Star Fight

ECE 4273

Professor: Dr. Erik Petrich

Authors: Giselle Chavez, Sam Musser

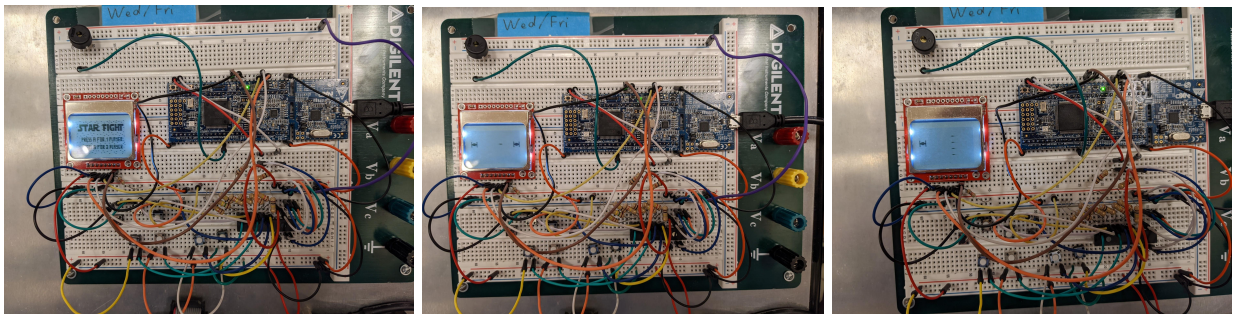
Not so long ago in a lab not very far, far way...

Star Fight Introduction Video: https://youtu.be/yauU1RXqylA

Star Fight Demo Video: https://youtu.be/rVxI7yN4u_g

General Description:

This blast from the past exciting hit of a game does not just take you to the retro 1980's, but to a galaxy far, far away. Star Fight delivers singleplayer and multiplayer action to players like you! Test your piloting skills as you dodge and weave through continuous laser fire in your single player adventure. Hone your dog-fighting capabilities as you go toe to toe with another player in the ultimate two player star-fighter face off! The fate of the Galaxy is in your hands. Main features include a display utilizing the Nokia 5110 graphical LCD with SPI interfacing to the subsystem of the MCUXpresso LPC1769, a custom controller with a serial interface utilizing the I2C subsystem of the LPC, an 18 note soundtrack, and technically advanced sound effects through the piezo and frequency manipulation. Prepare yourself for the ultimate experience…



Directions for Use:

Assuming the hardware has been built according to the specificity of the complete schematic found in the schematic section, all that has to be done is upload the software into the LPC1769. This can be done by creating a new semi-hosted C project that uses c99 in the MCUXpresso IDE. After uploading using a debug session, the microcontroller should run the program from the start every time it receives power. If the hardware is already built and the

software is already in the microcontroller, then by applying power the game begins! The gameplay controls are intuitive. After hearing the introductory theme and seeing the home screen, the user is prompted to press a button for either single player or two player. Pressing the corresponding button for a single player begins the endless runner gameplay. The buttons are organized by their respective actions. This means that the left three buttons are for the first player, just as that is the side of the screen the player appears on. The first player's buttons are positioned with one above, one below, and one in the middle that is slightly off vertically from the top and bottom. The top button moves the player up, and the bottom button moves the player down. Dodge the laser fire! The third, middle button comes into play if two players are selected.

When selecting multiplayer, the game pits two players against each other with the goal being to shoot the other player first while dodging their opponents laser fire. The buttons are the same as in single player. The left three are for player one, the left three are for player two, but the buttons in between the move up and move down options are used to trigger their lasers. In each mode, if a player is hit, the game is sent back to the home screen to allow for a different mode to be played. The game can be turned off by removing the power source, and if reapplied, the game will play the opening theme and present the home screen again just as it was before. For more in depth instructions as to the functionality and design of the project, the following design analysis offers explicit details on the hardware and software of the game.

Design Analysis:

Starting with the hardware, the LPC1769 will be utilizing the I2C and SPI subsystems. These subsystems are important because the controller for user input uses I2C interface while the graphical LCD uses SPI. As a quick review, I2C refers to an interface that generally speaking uses a start, stop, read, and write system so that the control in the group of systems knows what it needs to be reading or writing to, while also doing one object at a time due to a lack of availability for parallelism. The objects tethered to the I2C system share a clock, SCL, and SDA is where the data being read to or from goes into. SPI is similar in that it utilizes read or write, but for this case we only need write, known as MOSI. It utilizes a bus-out functionality with the clock, SCK, triggering every time a command is issued to the bus-out. Thus, specific pins need to be used in order to properly utilize these systems. For the I2C, p0.27 and p0.28 need to be used for the SDA and SCL respectively. For the SPI, p0.18 and p0.15 need to be used for the

MOSI and SCK. Again, MISO does not need to be used since data will only be sent out to the graphical LCD, not back into the microcontroller. The other registers on the LPC are arbitrarily chosen, but they have to be utilized in their GPIO modes and not as some restricted or off limits pin. For the graphical LCD, this includes reset, SCE, and the D/C input which chooses data or command mode. For the piezo, only a pin was used. However, in order to use this software specifically, the pins as seen in the complete schematic need to be used.

The I/O expander which uses I2C, needs all inputs on section GPA to be connected in parallel with switches and pull down resistors, as the switches will be connected to power for an active high input. In the schematic, they are connected on GPA7 to GPA0 with the following inputs in mind: player 1 up, player 1 down, player 1 shoot, player 2 up, player 2 down, player 2 shoot, single player, multiplayer. This is why the switches are physically placed as they are in the schematic, which is touched on in the directions for use. SDA and SCK need to be connected to the SDA and SCK on the microcontroller, with both lines having a separate resistor leading them to power. A0-2 needs to be grounded and reset needs to be high as it is active low.

The graphical LCD uses all of the corresponding pins mentioned in the LPC paragraph, but it also has leds that light up the screen. This is an optional function that really just depends on the work environment of the game, where it will be made or used. This project utilizes it with a 1 kilo-Ohm resistor taking it to power. The piezo also needs to be connected to ground, as the output goes from the LPC to the piezo.

With the class being Digital Design Lab, the software for the project has certain limitations. First limitation, written software cannot use any imported libraries. Second limitation, written software has to work directly with the registers of the LPC1769. Because of these restrictions, the source code does a lot to implement and create code that uses the subsystems of the microcontroller from scratch. This is not an issue, rather a clarification to be said when analyzing the code and seeing a lot of implementation of basic functions for the I2C and SPI subsystems. To best explain the software, the description will be categorized by the level of abstraction, working backwards as necessary to explain the gameplay, as seen in the final complete software section.The LPC1769 definitions needed are for the following: the GPIO's, the timer counter register, the I2C controls, the SPI subsystem, pin modes and selects, and finally power control for the I2C. After initializing all of the systems being used, the screen is cleared and the game begins.

The game method displays the home screen using the SPI write function to the GLCD, plays the opening game theme (The Imperial March), and then begins an infinite loop. This loop constantly checks for the user input using a check function. Check updates the user Input value to be the equivalent of the value delivered by the I/O expander to the LPC. If the value corresponds to what would be the single player input value, the endless runner begins. If the value corresponds to what would be the multiplayer value, the two player dog-fight begins. If single player mode is chosen, another loop begins that is dependent on the value of a variable chosen to be called gameOver. While this value is zero, the game loops continuously. Going through this single player loop, lasers are given a location to begin with which will come toward the player. The user input is then updated. They can move up or down depending on that value. The game is then updated, which moves the player if they move and also moves the lasers toward the player. Then there is a check for a hit. If the player is hit, gameOver becomes a 1 and the loop is broken. Outside the loop, gameOver is set back to zero so that another game may begin depending on the user selection. The multiplayer loop is very similar in that it depends on a value to continue the loop which only changes if there is a victory. It constantly checks for user inputs, updates values of the objects in the game if they move, updates the screen, then checks for wins. There are a lot more cases for inputs though, and this is to ensure that both players may push buttons at the same time.

Now to go deeper into the mechanics of these methods. The foundation of the game is a system of checks on arrays that make up the objects. The objects have a specific position on the rows and columns of bytes that make up the graphical LCD screen. When checking for hits, firing lasers, or even just determining where the lasers come from in singleplayer, this position is checked. The position is set as the leftmost byte of the object's image. Because of this, when checking for a hit on the right side of an object, the width of the object has to be incorporated in order to check for a hit. When firing a laser at an opponent, the position of the ship is checked so that the position can be written to the laser position. The lasers are unique in that they have an extra space in their array for validity. That is to say, if a laser is activated, that place in the array will be true. If the laser is not in use or reaches a boundary without hitting someone, it is set to false. Previously, it was mentioned that the endless runner checks the player position in order to decide where to fire lasers. This is because it is not truly randomized. At the time of creation, the team concluded that using a rand function constituted as importing and using a library. While the

team later found out this was acceptable, they came to a unique conclusion to create a seemingly randomized endless runner. The players position when the lasers hit the boundary determines where the next set of lasers come from. There are only four lasers shot at a time in the endless runner. With 6 rows of bytes to move across, this leaves two spaces for the player to choose from. Those two spaces have different sets of laser locations that will fire if chosen, and the process continues for each of the six rows the player could survive on.

The objects that make up Star Fight have to be constantly moved, updated, or checked. To do this, update methods were created to rewrite to the graphical LCD and update the positions as they are or as the player chooses. Everytime the screen updates, the output array, which holds the data for each of the 504 bytes in the graphical LCD, everything is cleared and set to all zeros in order to delete previous object positions. Then each object is checked and with the position, added to the same space in the array. While the width of the object is still positive, the loop keeps updating the object's position. If there is no more object to update, the loop moves to the next object. This only occurs with the lasers if they are active as there are checks to make sure they are before updating. A key part of updating the screen is actually moving the objects. The movement methods are called by the user within the main gameplay loop, unless they are automated in the updater like the lasers are. The user calls a movement function with a button press. The movement functions use cases to decide whether the user chose up or down, and then they go deeper into abstraction. These smaller methods do checks to make sure the object is not on the top most or bottom most row so that it does not return the object’s position as something that would otherwise not fit in the output array. This is similarly done with the lasers, which move horizontally. Their position is checked so that they do not cross the vertical boundaries of the screen. If they were to, because of how the graphical LCD works, they would just appear on the next row. This is because the graphical LCD outputs to the 84 bytes on the first row, then the second, and so on. The output is similar to reading a book. That is why, when referencing the code, the move up functions change array positions by an amount of 84 and why move left or right functions only move by an array position of 1. Moving an array position of 1 only moves the object’s pixels over by one pixel, but moving by 84 shifts an object up or down an entire row.

Now that there is a better understanding for how objects are updated and the game functions, these processes need to be analyzed with regard to the exact mechanics of how they are written or read by the microcontroller. This is better done by starting with the initialization

processes of the sub-systems. The I2C initialization mainly consists of setting the pin select options to the right values. Otherwise, the power control for the I2C clock has to be turned on, the pull up or down resistors for those specific pins have to be turned off, and then comes the pin selection. From there, the I2C bit frequency has to be set which is represented as follows:

$$pCLK(I2C)/10 = SCL = 1Mhz/10$$
$$therefore,\ the\ low\ divider\ and\ high\ divider = 5$$

Then the I2C is enabled by the control settings register. It is relatively short, but there is more to it. I2C requires serial functions in order to do anything. So from there the control settings have to be set to 1 at bits found in the complete software functions labeled under the category serial functions. What all of them have in common, the start, stop, read, and write, is they use the control clear and usually have to wait on the control settings to make sure things went through properly.

The SPI has to be allocated the proper bits in the pin selection, have certain outputs on registers to adjust for the graphical LCD, and then initialize the SPI in the LPC by adjusting the SPI control register. Then after setting the clock counter register to master mode, the graphical LCD can be initialized. The graphical LCD has to be cleared from previous use, but then it has to be put into command mode. Chip select is set to high and then to low as it is active low and begins the writing process. An array is created with the instructed values given by the Nokia 5110 GLCD data sheet, as seen in the GLCD_init() function in the software, and then that array is looped and written to the data register for the SPI subsystem. After every byte, though, is a check to make sure it went through properly, and it will wait in a loop until it is complete. Something to note that caused a lot of headache is the contrast setting on the GLCD. That value has to be tweaked depending on how the image is coming out, the voltage going in, and perhaps even what make of the GLCD is being used. After those values are sent to the GLCD, the LPC can then set it to data mode, so whatever data is sent to the data register of the SPI will now appear as pixels on the screen. This includes the home screen, and the game objects. The home screen, while visually stunning, is actually less complex than the gameplay objects. It is just an array of size 504 that is looped and output to the screen. Designing it was a process that

consisted of pixel editing, bitmap conversion, and array conversion. This is a similar process for the music, as the theme is just an array of the right amount of ticks necessary for the piezo.

The music in the game, at least how it is implemented, is reminiscent of the first project in this class. A square wave is output from a register which in turn creates a note on the piezo. This is done by using a while loop, that for a certain amount of seconds, consistently outputs high and low to the piezo for a designated amount of ticks. Ticks are the number of lines the code can go through at the operational frequency of the LPC. The amount of ticks that the signal is high or low determines the frequency of the note being played. In project 1, the equation that converts ticks to pulse width was found to be:

$$Pw = 0.002719(ticks) + 0.008410$$

Knowing that the duty cycle was to be 50% for maximum volume and the frequency of the notes to be used, it is just a matter of converting them to the pulse width and then into ticks for which a the process is as follows:

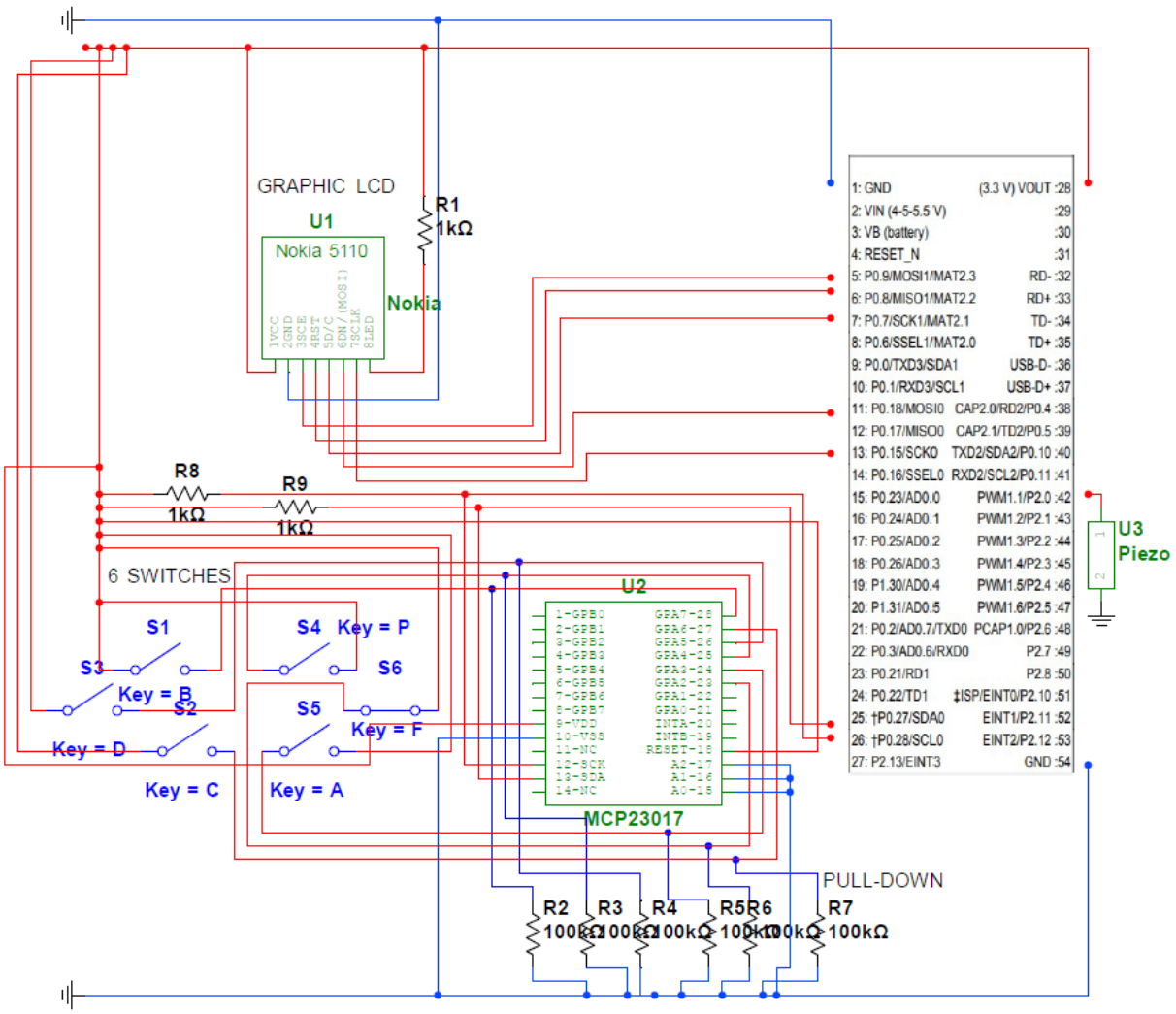
$$T = 1/f$$
$$Pw = Duty\ Cycle\ /\ T = 0.002719(ticks) + 0.008410$$

So knowing the notes of the Imperial March to consist of A, B, C, F, and E, their respective frequencies (110, 116, 65, 82, 87) can then be converted to the following ticks: 164, 155, 278, 220, 208. This also takes into account their octave. So playing the music goes through a for loop for each of the 18 notes, and it checks for the faster notes and plays them. If it is not one of the faster notes, then they are played as quarter notes. The other sound effects are just A and B notes that go off for a fired laser or a hit. All of this and more can be found in the complete final software section with notes and details surrounding all processes of the game with increasing levels of complexity as the reader goes further down the lines of code, which rely on more levels of abstraction.

Conclusion:

Overall, this project really became an exhibition and the culmination of all the subjects learned in this class. The project features frequency manipulation, serial interfaces, and an overall deeper understanding of the inner workings and subsystems of the LPC1769 microcontroller. On top of that, it really was a great learning experience which gave the team a newfound sense of enjoyment for embedded software. Despite current conditions like COVID-19, it really was an enjoyable and seemingly safe working environment that, despite the masks, shields, and anxiety, restored a slight sense of normalcy.

The Final Complete Schematic:



The Final Complete Software:

```
/*
===============================================================================
 Name        : ddlFinalProject.c
 Author      : $(Giselle Chavez and Sam Musser)
 Version     : 1.0
 Copyright   : $(copyright)
 Description : The main objective of this project is to create a
 vintage dog fighter game with an overt Star Wars theme. The main
 parts being used will include: LPC1769 for control, Nokia 5110 GLCD for
 display, piezo for music, and custom buttons with serial interface
 for user input.
===============================================================================
*/
#ifdef __USE_CMSIS
#include "LPC17xx.h"
#endif

#include <cr_section_macros.h>
//*****************************************************************************
//LPC1769 definitions
//

// I/O definitions for some of the outputs
#define FIO0DIR (*(volatile unsigned int *)0x2009c000)
#define FIO0PIN (*(volatile unsigned int *)0x2009c014)
#define FIO2DIR (*(volatile unsigned int *)0x2009c040)
#define FIO2PIN (*(volatile unsigned int *)0x2009c054)


//Timer Counter registers for wait function
#define T0TCR (*(volatile unsigned int *)0x40004004)  //control register
#define T0TC (*(volatile unsigned int *)0x40004008)   //status register

//I2C control definitions
#define I2C0CONSET (*(volatile unsigned int *)0x4001c000)
#define I2C0STAT (*(volatile unsigned int *)0x4001c004)
#define I2C0DAT (*(volatile unsigned int *)0x4001c008)
#define I2C0SCLH (*(volatile unsigned int *)0x4001c010)
#define I2C0SCLL (*(volatile unsigned int *)0x4001c014)
#define I2C0CONCLR (*(volatile unsigned int *)0x4001c018)

//SPI definitions
#define S0SPCR (*(volatile unsigned int *)0x40020000)  //control register
#define S0SPSR (*(volatile unsigned int *)0x40020004)  //status register
#define S0SPDR (*(volatile unsigned int *)0x40020008)  //data register
#define SPTCR (*(volatile unsigned int *)0x40020010)   //test register
#define S0SPCCR (*(volatile unsigned int *)0x4002000c) //clock counter reg
#define S0SPINT (*(volatile unsigned int *)0x4002001c) //interrupt register

//the pinmode definitions for the sclk and mosi
#define PINSEL0 (*(volatile unsigned int *)0x4002c000)
#define PINSEL1 (*(volatile unsigned int *)0x4002c004)
#define PINSEL4 (*( volatile unsigned int *)0x4002c010)
#define PINMODE1 (*(volatile unsigned int *)0x4002c044)

//power control to start the i2c power/control
#define PCONP (*( volatile unsigned int *)0x400fc0c4)

//*****************************************************************************
//wait function, rand, global constants, and variable definitions
//
//

//wait function that depends on 1MHz freq, useful so pixels do not shift
//at inconceivable speed or for input delay
void wait(float seconds)
{
    int start = T0TC;
    T0TCR |= (1<<0);
    seconds *= 1000000;
    while ((T0TC-start)< seconds) {}
}

//alternative wait function for the piezo
void tick(int ticks)
{
    volatile int cntDwn;
```

```
    for (cntDwn = 0; cntDwn < ticks; cntDwn++)
    {}
}

//variables
char output[504];      //array of the output bytes for the GLCD

//game object arrays will hold the following:
//initial arrayPosition (ie where they should go in output array)
//width (position will correspond to leftmost bit, so need to know right for bounds
//height (for the smaller objects that will be moving up by pixels instead of rows
int ball[] = {211, 2, 2};
int laser1[] = {250, 3, 1, 0};  //fourth laser value is to relay whether it is on or off
int laser11[] = {250, 3, 1, 0};
int laser2[] = {250, 3, 1, 0};
int laser21[] = {250, 3, 1, 0};
int tieFighter1[] = {169, 10, 8};
int tieFighter2[] = {241, 10, 8};

//sounds
//ticks in the alt wait function to deliver the imperial tune at desired frequencies
int imperialTune[] = {220, 220, 220, 278, 208, 220, 278, 208, 220,
            164, 164, 164, 155, 208, 220, 278, 208, 220};
int pewNoise[] = {164};
int hitNoise[] = {300};

//addresses for the I/O expander
int expWrite = 0x40;  //expander write address
int expRead = 0x41;   //expander read address
int DIRA = 0x00;      //Address for GPIOA
int GPIOA = 0x12;     //write to this then read for inputs
int GPPUA = 0x0C;     //This is to turn off pull up resistors on expander

int gameOver = 0;     //shows whether the game is on or lost

int ABRT;              //These variables are components of the status register
int MODF;              //may not be used for final iteration
int ROVR;
int WCOL;
int SPIF;

//user input value for the serial controller
volatile int inputVal = 0;

//tie shape
char tie[] = {0xFF, 0x18, 0x18, 0x3C, 0x3C, 0x3C, 0x3C, 0x18, 0x18, 0xFF};
//laser shape
char lzr[] = {0x08, 0x08, 0x08};
//ball shape, in case there is the desire to implement pong later
char bll[] = {0x18, 0x18};

//home screen mapping array (star fight)
char starFightBitMap [] = {
    0x04, 0x00, 0x10, 0x80, 0x00, 0x00, 0xE0, 0xE1, 0xE0, 0xE0,
    0xE0, 0xE0, 0xE0, 0xE4, 0xE0, 0xE0, 0xE0, 0xE2, 0xE0, 0xE0,
    0xE8, 0x00, 0x00, 0xE0, 0xE4, 0xE0, 0xE1, 0xE0, 0x00, 0x00,
    0xE0, 0xE0, 0xE0, 0x64, 0x60, 0x60, 0xE1, 0xE0, 0xC8, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x42, 0x00, 0xE0, 0xE0, 0xE0, 0xE8,
    0xE0, 0xE0, 0xE0, 0x00, 0xE2, 0xE0, 0xE0, 0x08, 0x81, 0xC0,
    0xE0, 0xE4, 0xE0, 0xE0, 0xE0, 0xE2, 0x00, 0xE0, 0xE0, 0xE4,
    0x01, 0x00, 0xE0, 0xE0, 0xE2, 0xE0, 0xE0, 0xE0, 0xE4, 0xE0,
    0xE0, 0xE0, 0x01, 0x00, 0x00, 0x02, 0x00, 0x70, 0x70, 0x70,
    0x73, 0x77, 0x7F, 0x7E, 0x3C, 0x00, 0x00, 0x00, 0x00, 0x7F,
    0x7F, 0x7F, 0x00, 0x00, 0x00, 0x70, 0x7C, 0x7F, 0x1F, 0x18,
    0x1F, 0x7F, 0x7C, 0x60, 0x7F, 0x7F, 0x7F, 0x0E, 0x1E, 0x3E,
    0x7E, 0x77, 0x63, 0x60, 0x61, 0x00, 0x00, 0x00, 0x00, 0x10,
    0x00, 0x7F, 0x7F, 0x7F, 0x0C, 0x0C, 0x00, 0x00, 0x7F, 0x7F,
    0x7F, 0x00, 0x1F, 0x3F, 0x7F, 0x78, 0x60, 0x6C, 0x7C, 0x7C,
    0x00, 0x7F, 0x7F, 0x7F, 0x06, 0x06, 0x7F, 0x7F, 0x7F, 0x00,
    0x7F, 0x7F, 0x7F, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x40,
    0x04, 0x00, 0x00, 0x00, 0x00, 0x20, 0x00, 0x00, 0x00, 0x02,
    0x00, 0x10, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00, 0x08, 0x00,
    0x00, 0x00, 0x00, 0x02, 0x00, 0x20, 0x00, 0x00, 0x04, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x08, 0x00, 0x00, 0x00, 0x00, 0x02, 0x20, 0x00, 0x00, 0x00,
    0x08, 0x00, 0x00, 0x20, 0x01, 0x00, 0x00, 0x00, 0x00, 0x02,
    0x00, 0x10, 0x00, 0x00, 0x00, 0x00, 0x10, 0x00, 0x00, 0x02,
    0x00, 0x00, 0x20, 0x00, 0x04, 0x00, 0x00, 0x10, 0x00, 0x01,
    0x40, 0x00, 0x00, 0x80, 0x00, 0x00, 0x00, 0x01, 0x80, 0x00,
    0x00, 0x3F, 0x05, 0x07, 0x00, 0x3F, 0x1D, 0x37, 0x00, 0x3F,
    0xA9, 0x29, 0x00, 0x27, 0x3D, 0x00, 0x27, 0x3D, 0x00, 0x00,
```

```
    0x00, 0x3F, 0x05, 0x05, 0x3F, 0x00, 0x00, 0x00, 0xBF, 0x05,
    0x05, 0x01, 0x00, 0x3F, 0x21, 0x3F, 0x00, 0x3F, 0x1D, 0x37,
    0x00, 0x00, 0x20, 0x00, 0x02, 0x3F, 0x00, 0x00, 0x00, 0x3F,
    0x05, 0x07, 0x00, 0xBF, 0x20, 0x00, 0x3F, 0x05, 0x3F, 0x00,
    0x27, 0x24, 0xBF, 0x00, 0x3F, 0x29, 0x29, 0x00, 0x3F, 0x1D,
    0x37, 0x00, 0x02, 0x20, 0x80, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x10, 0x00, 0x00, 0x00, 0x00, 0xF8, 0x28, 0x3A, 0x00, 0xF8,
    0xE8, 0xB8, 0x00, 0xF8, 0x48, 0x48, 0x00, 0x38, 0xE8, 0x00,
    0x38, 0xE9, 0x00, 0x00, 0x00, 0xF8, 0x48, 0x78, 0xC0, 0x10,
    0x00, 0x00, 0xF8, 0x28, 0x28, 0x08, 0x00, 0xFA, 0x08, 0xF8,
    0x00, 0xF8, 0xE8, 0xB8, 0x00, 0x01, 0x00, 0x98, 0xC8, 0x78,
    0x00, 0x00, 0x01, 0xF8, 0x68, 0x78, 0x00, 0xF8, 0x00, 0x00,
    0xF8, 0x2A, 0xF8, 0x00, 0x38, 0x20, 0xF8, 0x00, 0xF8, 0x48,
    0x48, 0x00, 0xF8, 0xE8, 0xB8, 0x00, 0x02, 0x40, 0x00, 0x04,
    0x10, 0x00, 0x02, 0x00, 0x00, 0x08, 0x40, 0x00, 0x04, 0x01,
    0x80, 0x00, 0x10, 0x01, 0x00, 0x01, 0x00, 0x11, 0x01, 0x01,
    0x40, 0x01, 0x09, 0x00, 0x01, 0x01, 0x20, 0x00, 0x04, 0x01,
    0x01, 0x81, 0x01, 0x08, 0x00, 0x00, 0x01, 0x08, 0x00, 0x00,
    0x00, 0x01, 0x41, 0x01, 0x00, 0x09, 0x00, 0x21, 0x00, 0x00,
    0x00, 0x01, 0x11, 0x01, 0x00, 0x00, 0x80, 0x01, 0x00, 0x08,
    0x00, 0x01, 0x01, 0x00, 0x21, 0x00, 0x01, 0x80, 0x11, 0x01,
    0x01, 0x00, 0x01, 0x01, 0x01, 0x08, 0x01, 0x00, 0x81, 0x20,
    0x00, 0x00, 0x08, 0x00
};



//initial game screen mapping array for reference (not quite to scale)
char gameScreen[] =  {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0xFF, 0x20, 0x70, 0x70, 0x70, 0x20, 0xFF, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xE0, 0xB0, 0xF0, 0xF0,
    0xE0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0xFF, 0x20, 0x70, 0x70, 0x70, 0x20, 0xFF, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x07, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x07, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x01, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x07, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x07, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00
};
```

```
//*********************************************************************
//standard initialization methods per LPC1769 and Nokia 5110 data sheets
//and the serial interface functions for the user input
//

//LPC I2C subsystem initialization
void I2C_init()
{
    //starts the I2C0 interface power/clock control
    PCONP |= (1<<7);

    //Need to turn off pull up or down resistors for p0.27/28
    PINMODE1 |= (1<<22) | (1<<24);

    //have to set p0.28/27 to SDA and SCL modes w/ 01
    //for bits 23/22 and 25/24
    PINSEL1 &= ~(1<<23); //p0.27
    PINSEL1 |= (1<<22);
    PINSEL1 &= ~(1<<25); //p0.28
    PINSEL1 |= (1<<24);

    //pclkI2C/10 = 1MHz/10= 100kHz = I2C bit frequency = SCL
    I2C0SCLL = 5;  //low div
    I2C0SCLH = 5;  //high div

    I2C0CONSET = (1<<6); //enable I2C
}

//LPC SPI subsystem initialization
void SPI_init()
{
    //Activates MOSI and SCLK modes by going high
    //at the following bits
    PINSEL0 |= (1<<31) | (1<<30);   //SCLK (clock, p0.15)
    PINSEL1 |= (1<<5) | (1<<4);     //MOSI (data out, p0.18)

    //setting the output pins (arbitrary, but we chose these)
    FIO0DIR |= (1<<9);   //GLCD SCE   output
    FIO0DIR |= (1<<8);   //GLCD RESET output
    FIO0DIR |= (1<<7);   //GLCD D/C   output (1/0)

    //initialize SPI subsystem in lpc
    S0SPCR &= ~(1<<2); //bit enabler
    S0SPCR &= ~(1<<3); //cphase as 0
    S0SPCR &= ~(1<<4); // cpolarity as 0 (sck active high)
    S0SPCR |= (1<<5);  //selects master mode (only one we need)
    S0SPCR &= ~(1<<6); //chooses least sig bit first if high
    S0SPCR &= ~(1<<7); //does not use interrupts

    //initialize the clock counter register to master mode
    S0SPCCR |= (1<<4); //sets number to 8
}

//Nokia 5110 GLCD initialization that ends in data mode (ready to write)
void GLCD_init()
{
    //initialize the GLCD display
    FIO0PIN |= (1<<8);
    FIO0PIN &= ~(1<<8);
    FIO0PIN |= (1<<8);   //clears the GLCD from previous use

    FIO0PIN &= ~(1<<7);  //D/C output start as low for command mode

    FIO0PIN |= (1<<9);   //chip select (active low)
    FIO0PIN &= ~(1<<9);   //falling edge begins the process

    output[0] = 0b00100001; //per instructions on GLCD data sheet
    output[1] = 0b11001000; //sets Vop to 16 x b[V] (per dataSheet) (contrast)
    output[2] = 0b00100000; //function set PD = 0 and V = 0 (normal instruction)
    output[3] = 0b00000100; //sets the temperature coefficient
    output[4] = 0b00010100; //makes the glcd bias mode 1:48
    output[5] = 0b00001100; //display control set normal mode (D=1, E=1)

    //writes these initialization commands to glcd in command mode
    for(int i = 0; i < 6; i++) {
        S0SPDR = output[i];

        //makes sure each byte passes through before moving on
        while (((S0SPSR >> 7)&1) == 0) {}
```

```
    }

    //puts glcd in data mode, now time for writing!
    FIO0PIN |= (1<<7); //data mode for the glcd
}

//Serial Functions:
//start function for beginning a read/write process
void start(void)
{
    I2C0CONSET = (1<<3);    //set SI
    I2C0CONSET = (1<<5);    //set STA
    I2C0CONCLR = (1<<3);    //clear SI
    //makes sure there is completion before moving on
    while((I2C0CONSET & (1<<3)) == 0) {}
    I2C0CONCLR = (1<<5);    //clear STA
}

//read function for reading in data to the i2c
int read(int heard)
{
    if(heard) {
        I2C0CONSET = (1<<2); //accepts data
    } else {
        I2C0CONCLR = (1<<2);
    }
    I2C0CONCLR = (1<<3);
    //waits for complete
    while((I2C0CONSET & (1<<3)) == 0) {}
    return (I2C0DAT & 0xFF);
}

//write function for i2c to output bit values
void write(int num)
{
    I2C0DAT = num;      //takes in the information
    I2C0CONCLR = (1<<3);
    //waits for completion
    while((I2C0CONSET & (1<<3)) == 0) {}
}

//stop function for ending the read/write process
void stop(void)
{
    I2C0CONSET = (1<<4);    //sets the sto
    I2C0CONCLR = (1<<3);
    //same idea as in the start function
    while((I2C0CONSET & (1<<4)) != 0) {}
}


//*************************************************************************
//the object movement functions
//
//
/*
"shift" moves objects by the pixel on the glcd whereas
"move" moves objects by the byte
*/

//moves up 1 pixel
void shiftUp() {}

//moves down 1 pixel
void shiftDown() {}

//moves up 8 pixels
int moveUp(int arrayNum)
{
    //makes sure cannot move past upper bounds
    if (arrayNum > 83) {
        arrayNum -= 84;
    }

    return arrayNum;
}

//moves down 8 pixels
int moveDown(int arrayNum)
{
    //makes sure cannot move past lower bounds
```

```
    if (arrayNum < 419) {
        arrayNum += 84;
    }

    return arrayNum;
}

//moves right by 1 pixel
int shiftRight(int arrayNum, int width)
{
    //takes into account the rightmost pixel using width
    int arrayNumR = arrayNum + width;

    //makes sure it cannot move onto the next row
    if((arrayNumR != 83) && (arrayNumR != 167) && (arrayNumR != 251) && (arrayNumR !=
            335) && (arrayNumR != 419) && (arrayNumR != 503)) {
        arrayNumR += 1;
    }

    arrayNum = arrayNumR - width;

    return arrayNum;
}

//moves left by 1 pixel
int shiftLeft(int arrayNumL)
{
    //makes sure it cannot move into the previous row
    if((arrayNumL != 0) && (arrayNumL != 84) && (arrayNumL != 168) && (arrayNumL !=
            252) && (arrayNumL != 336) && (arrayNumL != 420)) {
        arrayNumL -= 1;
    }

    return arrayNumL;
}

//tie1 movement method
void tie1Move(int joy)
{
    switch(joy) {
        case 0:
            tieFighter1[0] = moveUp(tieFighter1[0]);
            break;
        case 1:
            tieFighter1[0] = moveDown(tieFighter1[0]);
            break;
    }
}

//tie2 movement method
void tie2Move(int stick)
{
    switch(stick) {
        case 0:
            tieFighter2[0] = moveUp(tieFighter2[0]);
            break;
        case 1:
            tieFighter2[0] = moveDown(tieFighter2[0]);
            break;
    }
}

//would utilize angle and direction tracker and shift vertical/horizontal for diagonal
void ballMove()
{}

//triggers at laser button activation, sets current position, and true for laser
//to appear on the screen
void fireLaser(int player)
{
    switch(player) {
        case(1):
            laser1[0] = tieFighter1[0] + 11; //sets laser initial position
            laser1[3] = 1;                   //sets laser as active
            break;

        case(2):
            laser2[0] = tieFighter2[0] - 2; //sets laser initial position
            laser2[3] = 1;                  //sets laser as true
            break;
```

```
        case(3):
            laser11[0] = tieFighter1[0] + 11; //sets laser initial position
            laser11[3] = 1;                   //sets laser as true
            break;

        case(4):
            laser21[0] = tieFighter2[0] - 2; //sets laser initial position
            laser21[3] = 1;                  //sets laser as true
            break;
    }
}

//*****************************************************************************
//screen functions such as reset, clear, or updates and
//also the user input checker

//resets the game's objects' locations in case of win/lose (single player)
void reset()
{
    ball[0] = 211;
    ball[1] = 2;
    ball[2] = 2;
    laser1[0] = 250;
    laser1[1] = 3;
    laser1[2] = 1;
    laser1[3] = 0;
    laser2[0] = 250;
    laser2[1] = 3;
    laser2[2] = 1;
    laser2[3] = 0;
    laser11[0] = 250;
    laser11[1] = 3;
    laser11[2] = 1;
    laser11[3] = 0;
    laser21[0] = 250;
    laser21[1] = 3;
    laser21[2] = 1;
    laser21[3] = 0;
    tieFighter1[0] = 169;
    tieFighter1[1] = 10;
    tieFighter1[2] = 8;
    tieFighter2[0] = 241;
    tieFighter2[1] = 10;
    tieFighter2[2] = 8;
}

//displays outputs current values to the screen
void updateScreen()
{
    for (int f = 0; f < 504; f++) {
        S0SPDR = output[f];

        while (((S0SPSR >> 7)&1) == 0) {}
    }
}

//sets all output values to that of what "would" be a blank screen
void clrOutput()
{
    for (int m = 0; m < 504; m++) {
        output[m] = 0x00;
    }
}

//sets all output values to zero then displays the blank screen
void clrScreen()
{
    clrOutput();

    updateScreen();
}

//outputs the home screen to the display
void displayHome()
{
    for (int hh = 0; hh < 504; hh++) {
        S0SPDR = starFightBitMap[hh];

        while (((S0SPSR >> 7)&1) == 0) {}
    }
}
```

```
//updates the screen with current object positions (single player)
void updateSingleGame()
{
    //clears output to erase previous object positions
    //before rewriting them
    clrOutput();

    //to keep tabs on the object array values
    int temp;
    temp = 0;

    //sets the first tie fighter position in output
    for (int e = tieFighter1[0]; e < (tieFighter1[0] + 10); e++) {
        output[e] = tie[temp];
        temp++;
    }

    temp = 0;

    //sets the new laser positions if they are on and updates them
    if (laser1[3] == 1) {
        for (int lr = laser1[0]; lr < laser1[0] + 3; lr++) {
            output[lr] = lzr[temp];
            temp++;
        }

        laser1[0] = shiftLeft(laser1[0]);
    }

    temp = 0;

    //sets the new laser positions if they are on and updates them
    if (laser2[3] == 1) {
        for (int y = laser2[0]; y < laser2[0] + 3; y++) {
            output[y] = lzr[temp];
            temp++;
        }

        laser2[0] = shiftLeft(laser2[0]);
    }

    temp = 0;

        //sets the new laser positions if they are on and updates them
        if (laser11[3] == 1) {
            for (int b = laser11[0]; b < laser11[0] + 3; b++) {
                output[b] = lzr[temp];
                temp++;
            }

            laser11[0] = shiftLeft(laser11[0]);
        }

        temp = 0;

        //sets the new laser positions if they are on and updates them
        if (laser21[3] == 1) {
            for (int q = laser21[0]; q < laser21[0] + 3; q++) {
                output[q] = lzr[temp];
                temp++;
            }

            laser21[0] = shiftLeft(laser21[0]);
        }


    //updates the screen with these positions
    updateScreen();


}

//updates the screen with current object positions (multiplayer)
void updateMultGame()
{
    //clears output to erase previous object positions
    //before rewriting them
    clrOutput();

    //to keep tabs on the object array values
```

```
    int temp;
    temp = 0;

    //sets the first tie fighter position in output
    for (int g = tieFighter1[0]; g < (tieFighter1[0] + 10); g++) {
        output[g] = tie[temp];
        temp++;
    }

    temp = 0;

    //sets the second tie fighter position in output
    for (int y = tieFighter2[0]; y < (tieFighter2[0] + 10); y++) {
        output[y] = tie[temp];
        temp++;
    }

    temp = 0;

    //sets the new laser positions if they are on and updates them
    if (laser1[3] == 1) {
        for (int lr = laser1[0]; lr < laser1[0] + 3; lr++) {
            output[lr] = lzr[temp];
            temp++;
        }

        laser1[0] = shiftRight(laser1[0], laser1[1]);
    }

    temp = 0;

    //sets the new laser positions if they are on and updates them
    if (laser2[3] == 1) {
        for (int le = laser2[0]; le < laser2[0] + 3; le++) {
            output[le] = lzr[temp];
            temp++;
        }

        laser2[0] = shiftLeft(laser2[0]);
    }

    //sets the new laser positions if they are on and updates them
        if (laser11[3] == 1) {
            for (int lr = laser11[0]; lr < laser11[0] + 3; lr++) {
                output[lr] = lzr[temp];
                temp++;
            }

            laser11[0] = shiftRight(laser11[0], laser11[1]);
        }

        temp = 0;

        //sets the new laser positions if they are on and updates them
        if (laser21[3] == 1) {
            for (int le = laser21[0]; le < laser21[0] + 3; le++) {
                output[le] = lzr[temp];
                temp++;
            }

            laser21[0] = shiftLeft(laser21[0]);
        }

    //updates the screen with these positions
    updateScreen();
}

//*****************************************************************************
//Final game functions and music:
//checks for presses, wins, fire laser, noise output, and game loop

//checks the data values read from the I/O expander (serial input)
//and sets it equal to the input value
void checkIn()
{
    start();
    write(expWrite);
    write(GPIOA);
    stop();

    start();
```

```
    write(expRead);
    inputVal = read(0);
    stop();
}

//plays imperial theme
void playTheme()
{
    for (int note = 0; note < 18; note++)
    {
         int start = T0TC;
         T0TCR |= (1<<0);
         if ((note == 4) || (note == 7) || (note == 13) || (note ==16) ||
                 (note == 3) || (note == 6) || (note == 12) || (note ==15))
         {
             while ((T0TC-start)< 200000) {
                         FIO2PIN |= (1<<0);
                         tick(imperialTune[note]);
                         FIO2PIN &= ~(1<<0);
                         tick(imperialTune[note]);
                         }
                    wait(0.05);
         }
         else{
         while ((T0TC-start)< 400000) {
            FIO2PIN |= (1<<0);
            tick(imperialTune[note]);
            FIO2PIN &= ~(1<<0);
            tick(imperialTune[note]);
            }
         wait(0.1);
         }
    }
}

//plays laser noise
void pewPew()
{
       int start = T0TC;
       T0TCR |= (1<<0);
       while ((T0TC-start)< 100000)
       {
           FIO2PIN |= (1<<0);
           tick(pewNoise[0]);
           FIO2PIN &= ~(1<<0);
           tick(pewNoise[0]);
       }
}

//
void targetHit()
{
       int start = T0TC;
       T0TCR |= (1<<0);
       while ((T0TC-start)< 100000)
       {
           FIO2PIN |= (1<<0);
           tick(hitNoise[0]);
           FIO2PIN &= ~(1<<0);
           tick(hitNoise[0]);
       }
}

//positions lasers to output almost randomly for single player laser dodge
void comeAtMeBro()
{
    //generates random number between 1 and 6 to output a laser to
    //one of the rows
    //Also lasers based on current player position
    //Possible laser positions: 80 164 248 332 416 500
    if ((!laser1[3]) && (tieFighter1[0] == 1))
        {
            pewPew();
            pewPew();
            laser1[0] = 80;
            laser11[0] = 164;
            laser2[0] = 332;
            laser21[0] = 416;

            laser1[3] = 1;
            laser11[3] = 1;
```

```
            laser2[3] = 1;
            laser21[3] = 1;
        }
    else if ((!laser1[3]) && (tieFighter1[0] == 85))
    {
        pewPew();
        pewPew();
        laser1[0] = 80;
        laser11[0] = 164;
        laser2[0] = 248;
        laser21[0] = 500;

        laser1[3] = 1;
                    laser11[3] = 1;
                    laser2[3] = 1;
                    laser21[3] = 1;
    }

    else if ((!laser1[3]) && (tieFighter1[0] == 169))
    {
        pewPew();
        pewPew();
        laser1[0] = 248;
        laser11[0] = 164;
        laser2[0] = 332;
        laser21[0] = 416;

        laser1[3] = 1;
                    laser11[3] = 1;
                    laser2[3] = 1;
                    laser21[3] = 1;
    }

    else if ((!laser1[3]) && (tieFighter1[0] == 253))
    {
        pewPew();
        pewPew();
        laser1[0] = 80;
        laser11[0] = 332;
        laser2[0] = 500;
        laser21[0] = 416;

        laser1[3] = 1;
                    laser11[3] = 1;
                    laser2[3] = 1;
                    laser21[3] = 1;
    }

    else if ((!laser1[3]) && (tieFighter1[0] == 337))
    {
        pewPew();
        pewPew();
        laser1[0] = 248;
        laser11[0] = 164;
        laser2[0] = 500;
        laser21[0] = 416;

        laser1[3] = 1;
                    laser11[3] = 1;
                    laser2[3] = 1;
                    laser21[3] = 1;
    }

    else if ((!laser1[3]) && (tieFighter1[0] == 421))
            {
        pewPew();
        pewPew();
                laser1[0] = 80;
                laser11[0] = 164;
                laser2[0] = 332;
                laser21[0] = 500;

                laser1[3] = 1;
                            laser11[3] = 1;
                            laser2[3] = 1;
                            laser21[3] = 1;
            }
}

//checks positions of objects for hits/wins
//and also clears lasers if they have reached bounds without a hit
```

```
void gameOverSingle()
{
    //ahhh you've been shot! orrrr you won! good job.
    if((laser1[0] == (tieFighter1[0] + tieFighter1[1])) || (laser11[0] == (tieFighter1[0] + tieFighter1[1])) ||
            (laser2[0] == (tieFighter1[0] + tieFighter1[1])) || (laser21[0] == (tieFighter1[0] + tieFighter1[1])))
{
        targetHit();
        wait(2);
        reset();
        displayHome();
        gameOver = 1;

    }

    //all cases where lasers avoid ship and need to be reset
    if (((laser1[0]) != tieFighter1[0]) && ((laser1[0] == 11) ||
                                            (laser1[0] == 95) || (laser1[0] == 179) ||
                                            (laser1[0] == 263) || (laser1[0] == 347) ||
                                            (laser1[0] == 431))) {
            laser1[3] = 0;
            laser1[0] = tieFighter2[0];
        }

    if (((laser11[0]) != tieFighter1[0]) && ((laser11[0] == 11) ||
                                                (laser11[0] == 95) || (laser11[0] == 179) ||
                                                (laser11[0] == 263) || (laser11[0] == 347) ||
                                                (laser11[0] == 431))) {
                laser11[3] = 0;
                laser11[0] = tieFighter2[0];
            }

    if (((laser2[0]) != tieFighter1[0]) && ((laser2[0] == 11) ||
                                            (laser2[0] == 95) || (laser2[0] == 179) ||
                                            (laser2[0] == 263) || (laser2[0] == 347) ||
                                            (laser2[0] == 431))) {
            laser2[3] = 0;
            laser2[0] = tieFighter2[0];
        }

    if (((laser21[0]) != tieFighter1[0]) && ((laser21[0] == 11) ||
                                            (laser21[0] == 95) || (laser21[0] == 179) ||
                                            (laser21[0] == 263) || (laser21[0] == 347) ||
                                            (laser21[0] == 431))) {
            laser21[3] = 0;
            laser2[0] = tieFighter2[0];
        }
}

void gameOverMult()
{
    //player 1 win
    if ((laser1[0] + 2 == tieFighter2[0]) || (laser11[0] + 2 == tieFighter2[0])) {
        targetHit();
        wait(2);
        reset();
        displayHome();
        gameOver = 1;
        return;
    }

    //player 2 win
    if ((laser2[0] == (tieFighter1[0] + 10)) || (laser21[0] == (tieFighter1[0]+10))) {
        targetHit();
        wait(2);
        reset();
        displayHome();
        gameOver = 1;
        return;
    }


    //all cases in which the laser's avoid the ships and need to be reset
    //
    //player 1 laser 1
    if (((laser1[0] + 2) != tieFighter2[0]) && (((laser1[0] + 2) == 73) ||
            ((laser1[0] + 2) == 157) || ((laser1[0] + 2) == 241) ||
            ((laser1[0] + 2) == 325) || ((laser1[0] + 2) == 409) ||
            ((laser1[0] + 2) == 493))) {
        laser1[3] = 0;
        laser1[0] = tieFighter1[0];
    }
```

```
    //player 1 laser 2 (laser11)
    if (((laser11[0] + 2) != tieFighter2[0]) && (((laser11[0] + 2) == 73) ||
            ((laser11[0] + 2) == 157) || ((laser11[0] + 2) == 241) ||
            ((laser11[0] + 2) == 325) || ((laser11[0] + 2) == 409) ||
            ((laser11[0] + 2) == 493))) {
        laser11[3] = 0;
        laser11[0] = tieFighter1[0];
    }

    //player 2 laser 1
    if (((laser2[0]) != tieFighter1[0]) && ((laser2[0] == 11) ||
                                            (laser2[0] == 95) || (laser2[0] == 179) ||
                                            (laser2[0] == 263) || (laser2[0] == 347) ||
                                            (laser2[0] == 431))) {
        laser2[3] = 0;
        laser2[0] = tieFighter2[0];
    }

    //player 2 laser 2 (21)
    if (((laser21[0]) != tieFighter1[0]) && ((laser21[0] == 11) ||
            (laser21[0] == 95) || (laser21[0] == 179) ||
            (laser21[0] == 263) || (laser21[0] == 347) ||
            (laser21[0] == 431))) {
        laser21[3] = 0;
        laser21[0] = tieFighter2[0];
    }
}

//let the game begin!!
void playGame()
{
    displayHome();
    playTheme();
    while(1) {
        checkIn();

        //single player game loop
        if (inputVal == 2) {
            while(!gameOver) {
                comeAtMeBro();

                checkIn();                      //user input

                if (inputVal == 128) {          //up button
                    tie1Move(0);
                } else if (inputVal == 64) {    //down button
                    tie1Move(1);
                }
                updateSingleGame();             //updates positions and screen
                gameOverSingle();               //checks for loss
            }
        }
        gameOver = 0;                           //resets value for replay

        //multiplayer game loop
        if (inputVal == 1) {
            while(!gameOver) {
                checkIn();

                if ((inputVal == 128) || (inputVal == (128+16)) ||
                        (inputVal == (128+8)) || (inputVal == (128+4))) {
                    tie1Move(0);
                } else if ((inputVal == 64) || (inputVal == (64+16)) ||
                        (inputVal == (64+8)) || (inputVal == (64+4))) {
                    tie1Move(1);
                }
                checkIn();
                if ((inputVal == 32) || (inputVal == (32+16)) ||
                        (inputVal == (32+8)) || (inputVal == (32+4)))
                {         //fire laser
                    if (laser1[3]) {          //fire second laser if first is
                        fireLaser(3);         //active
                        pewPew();
                    } else {
                        fireLaser(1);
                        pewPew();
                    }
                }
                checkIn();
                if ((inputVal == 16) || (inputVal == (128+16)) ||
```

```
                        (inputVal == (16+64)) || (inputVal == (16+32)))
                {
                    tie2Move(0);
                } else if ((inputVal == 8) || (inputVal == (128+8)) ||
                        (inputVal == (8+64)) || (inputVal == (8+32)))
                {
                    tie2Move(1);
                }
                checkIn();
                if ((inputVal == 4) || (inputVal == (4+16)) ||
                        (inputVal == (4+64)) || (inputVal == (4+32)))
                {         //player 2 lasers
                    if (laser2[3])
                    {
                        fireLaser(4);
                        pewPew();
                    }
                    else
                    {
                        fireLaser(2);
                        pewPew();
                    }
                }
                updateMultGame();
                gameOverMult();
            }
        }
        gameOver = 0;
    }
}

//main method which deploys initialization functions, sets up read inputs,
//and begins the game!
int main(void)
{
    //I2C initialization for the user input
    I2C_init();

    //SPI initialization for the subsystem in the LPC1769
    SPI_init();

    //GLCD initialization, ready for writing
    GLCD_init();

    //activates P2.0 as output for piezzo (music initialization)
    PINSEL4 &= ~(1<<1);
    PINSEL4 &= ~(1<<0);
    FIO2DIR |= (1<<0);

    //prepare the I/O expander for reading input
    start();
    write(expWrite);
    write(DIRA);
    write(0xFF); //write 1's to DIRA to activate as input pins
    stop();

    start();
    write(expWrite);
    write(GPPUA);
    write(0x00); //write 0's to GPPUA to turn off pull up resistors;
    stop();

    clrScreen();

    //let the battle begin!
    playGame();

}
```