Serial Terminal

Design Documentation

William Hatch and Scott Sorensen

April 2013

Table of Contents

1 Introduction

This document describes the design of a serial computer terminal. The microcontroller reads input from a keyboard connected to the PS/2 port, outputs the input received via the RS-232 to the host computer, and then receives data from the host computer to output to the LCD screen.

2 Scope

In this document is shown how to connect a microcontroller and its components to a personal computer and the software design necessary for the communication between all of the devices.

This document does not cover the host computer's software nor the mechanical design of the components.

3 Design Overview

3.1 Requirements

1. The system will receive data from a PS/2 keyboard.

2. The system will be able to translate the key codes from the PS/2 keyboard to usable ASCII characters for the host computer.

3. The system will transmit and receive data to the host computer via USART

4. The system will be able to handle color codes and ANSI escape sequences.

5. The system will print the correct characters and colors to the LCD screen

6. The system will print a cursor to the LCD screen

3.2 Theory of Operation

Serial terminals are used to transfer data between a computer system and its users. The user types at the terminal's keyboard, and the ascii codes are sent to the computer to be interpreted. The computer outputs ascii letters and control sequences to the terminal, which are then displayed, sounded as terminal beep alerts, or used to alter the state of the display.

There are many different terminal designs used on various systems ranging from UNIX mainframes to DOS PCs. The capabilities and features of serial terminals vary widely, and the control code standards are complicated. The terminal in this design has a large subset of capabilities of the virtual Linux console, and is largely compatible with vt100 series terminals. Control codes supported include cursor manipulation, visual effects, terminal state saving, and basic line editing and output codes.

4. Design Details

4.1 Hardware

The following hardware components are used for this design:

1. STM32F103RC Micro-controller
2. SSD 1289 LCD Screen
3. RS-232 Module
4. Micro-controller Onboard DAC
5. Speaker
6. PS/2 Keyboard
7. Serial Cable
8. Host Computing System

A schematic of the design is included in Appendix A.

4.2 Software Design

The software for the design consists of the configuration of the devices used, as well as the handling of receiving, transmitting, and interpreting data between all of the devices.

The main function calls various initialization functions, along with a function to clear the screen. It then goes into an infinite while loop and functions for handling the usart data, handling the ps2 data, and refreshing the screen are continuously called.

A software flow chart is included in Appendix B.

4.2.1 PS/2 Keyboard Communication

The PS/2 port on the micro-controller shares pins with PC3 and PC4, which are the clock and data for the PS/2 Keyboard, respectively. The pins are configured as pull-up/pull-down input as the micro-controller will not be sending data to the keyboard. The external interrupt is enabled for the pins as well as the AFIO clock.

The PS/2 interrupt has various static variables to handle the start bit, the stop bit, and the data bits. The interrupt has an infinite while loop which checks which bit the has been received, and exits out of the loop depending on which bits have been received so far. Once a start bit and 8 data bits have been received, the data received is interpreted to be a special or regular byte. If it isn't a special byte, the code is translated to an ASCII code via a map so the host computer can read the data properly. It is then put into a data buffer. There are various special bytes that need to be handled differently: the shift key code, the ctrl key code, the alt key code, and the code signifying an up keystroke. None of these are put into the data buffer. For the shift and ctrl codes, modified maps are used to pass different ASCII values. For the alt key code, the ASCII escape code is put into the buffer to precede whatever key is being pressed down with it. If the byte is 0xF0, it signifies an upstroke of a key, and a variable is set to true to handle the next byte read. If the byte following an upstroke is shift or ctrl, then the data being translated goes back to using the default map, and for alt the escape code is no longer sent.

4.2.2 USART2 Communication

The data from the PS/2 keyboard is put into a buffer which is then sent transmitted via the USART2. In the USART2 initialize function the AFIO clock and alternate function for GPIOA are set, along with an interrupt enable to be triggered when the rx (receive) buffer is no longer empty. The tx function transmits the data from the PS/2 buffer to the host computer. The USART2 interrupt handles the data received from the host computer and puts it into a buffer which is processed to handle control codes and write the necessary data to the LCD screen.

4.2.3 LCD

The LCD screen is configured to use the correct pins from GPIOA and GPIOC, and it is initialized to have a horizontal configuration. Functions are written for clearing the screen, writing to the screen, setting the position of the cursor, writing the commands and data for the LCD, and drawing lines of characters to the screen.

An internal buffer of screen data is kept, with the characters to be drawn at each row and column, along with their individual visual effects. When they are printed their row and column positions are translated into x and y coordinates on the screen, and a bitmap is retrieved and drawn for each ascii character. The printing function displays the bitmap with appropriate foreground and background colors, as well as other effects such as underlining, one line of pixels at a time to the 8x16 character space. As the screen is periodically refreshed, characters set to blink are toggled on and off to produce the effect.

4.2.4 Control Codes and Escape Sequences

While most ascii characters received by the terminal are drawn to the screen, some must be handled as control codes or escape sequences that perform special functions. There is a state machine function that handles all the characters received from the host computer. Basic control codes are a single character and have simple functions, such as a carriage return, line feed, or bell (alert) character. Simple escape sequences start with an escape (ESC) character, then a character indicating a function, usually slightly more complex than the single character control codes. As terminal capabilities rapidly expanded at the time of creating these standards, some of them became somewhat complex, and to handle the more advanced features of graphical terminals, the CSI, or Control Sequence Inductor code standards were made. CSI codes begin with either a single CSI control character or an escape character followed by a left bracket (ESC[). They then have a sequence of ascii decimal numbers separated with semicolons. Finally they are terminated by a single character. The ending character determines how the numbers are interpreted and handled. This provides complicated functions for adding color and other visual effects, arbitrary cursor movement, and other advanced functions.

4.2.5 Visual Effects

The visual effects used for the terminal include adding background color, foreground color, underline for the text, bold (brightly colored) text, and blinking text. The various CSI codes are passed from the host computer and each code is interpreted to display correctly to the screen.

4.2.6 Onboard DAC

The DAC uses a timer interrupt to play sound according to a 40 entry wave table whenever a bell character is received from the host.

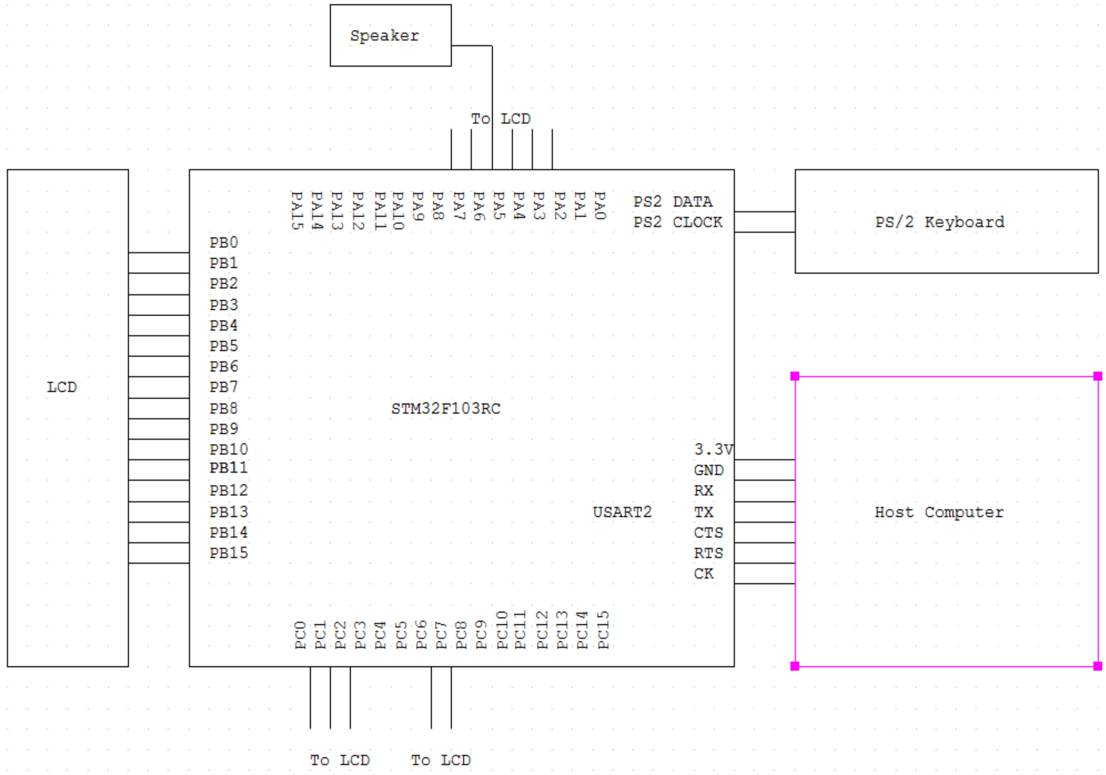
5. Verification

The system was hand tested to see that it received, transferred, and displayed data properly. A color comparison script was used to display all possible color combinations, and an effects script was written to test blinking, underlining, and reverse-video effects. Cursor manipulation was tested by viewing compatibility with programs such as Vim which use Curses libraries to manipulate the cursor and modify arbitrary sections of the display. The design is verified to work properly, although many programs are not designed to handle a display size as small as 15 rows x 40 columns, and such programs display poorly. Programs written to scale to very small displays display perfectly, however.

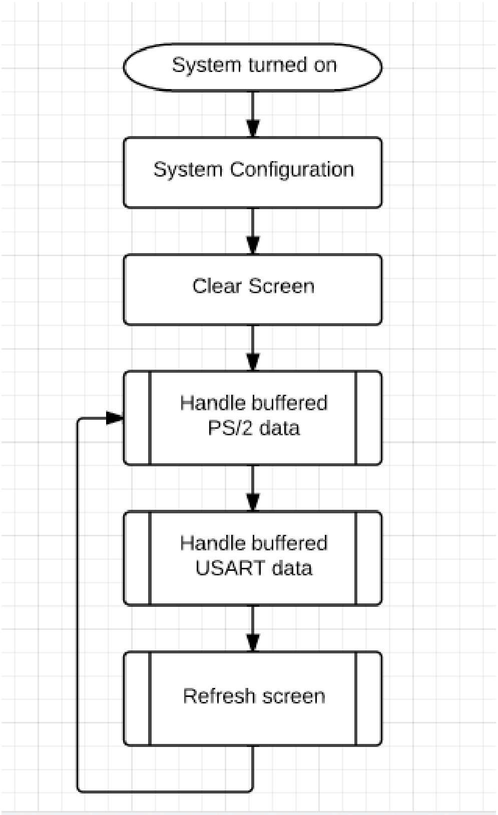
6. Conclusion

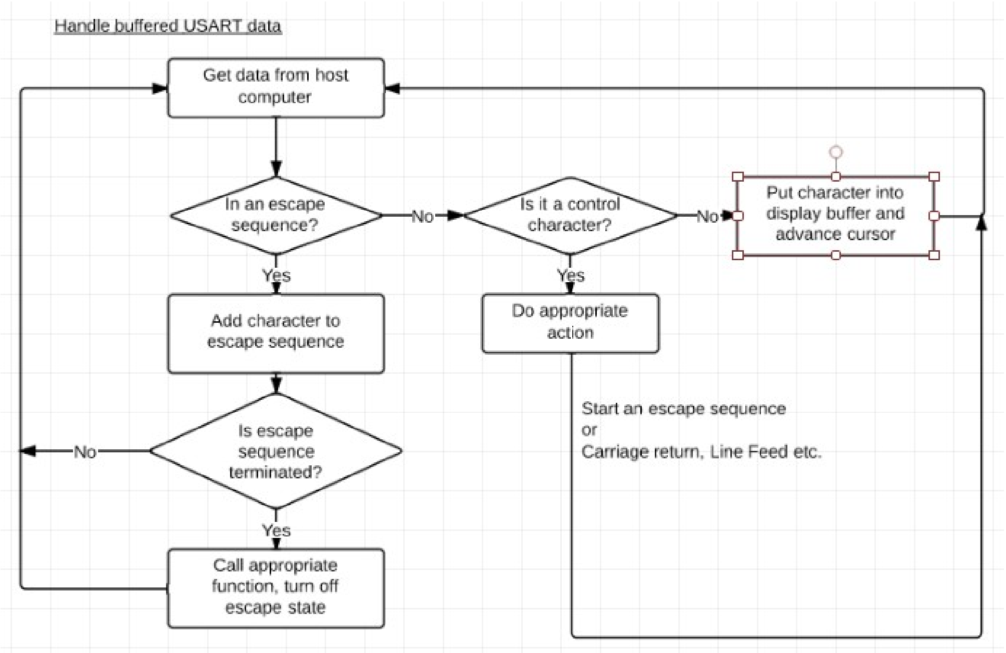
Serial terminals are an important part of computer history, and remain in use even today for various server administration tasks. The terminal outlined in this document is a functional, small, and light design which would be useful. The main drawback is the restrictive screen size. If a larger LCD screen were used to increase the display area it would be a useful terminal for actual use by people who use serial terminals today. For most practical applications, however, a software terminal emulator is recommended.

Appendix A: Hardware Schematic

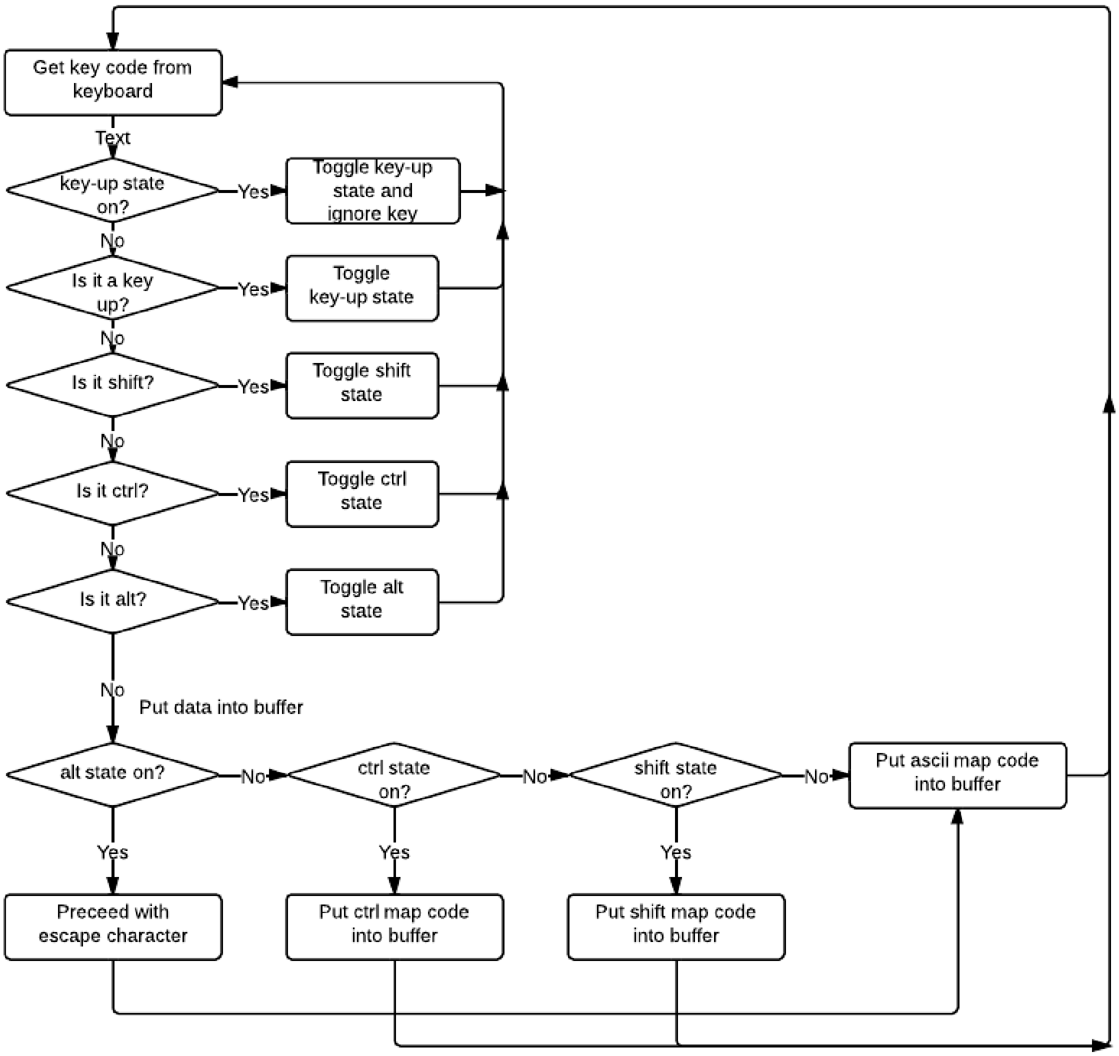


Appendix B: Software Flow Charts



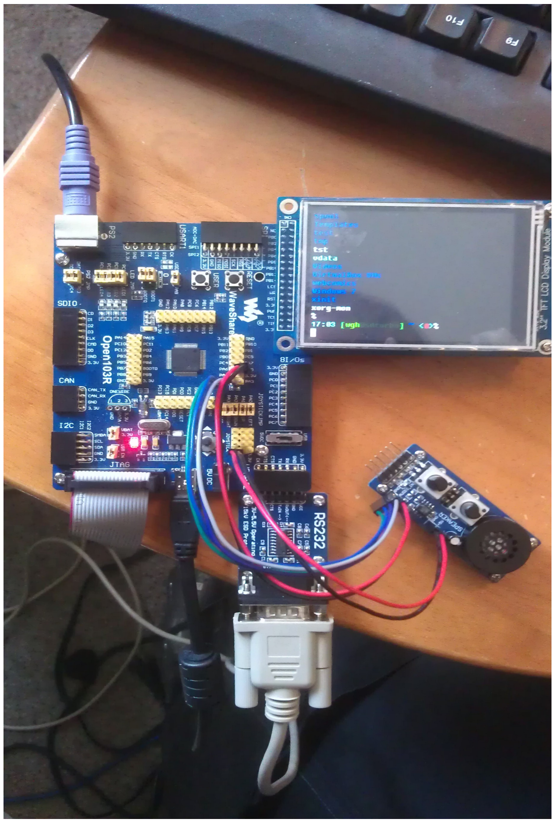

Handle buffered USART data
Get data from host computer
In an escape sequence?
No
Is it a control character?
No
Put character into display buffer and advance cursor
Yes
Add character to escape sequence
Yes
Do appropriate action
Start an escape sequence
or
Carriage return, Line Feed etc.
No
Is escape sequence terminated?
Yes
Call appropriate function, turn off escape state

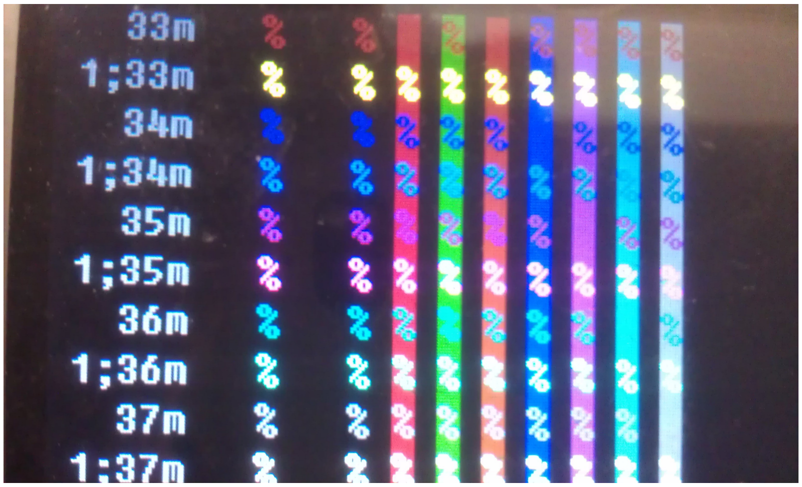
Handle buffered PS/2 data



Appendix C: Verification Photographs

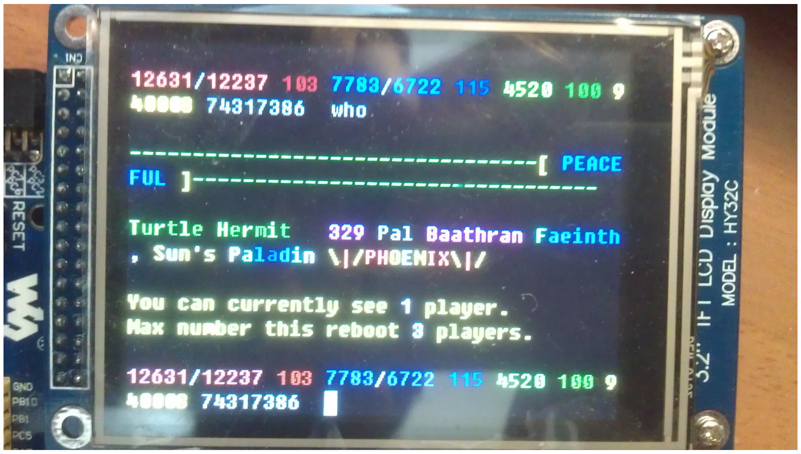
View of hardware setup, connected to Linux computer, and color test script output.





```
   33m  %   %  %  %  %  %  %  %  %
 1;33m  %   %  %  %  %  %  %  %  %
   34m  %   %  %  %  %  %  %  %  %
 1;34m  %   %  %  %  %  %  %  %  %
   35m  %   %  %  %  %  %  %  %  %
 1;35m  %   %  %  %  %  %  %  %  %
   36m  %   %  %  %  %  %  %  %  %
 1;36m  %   %  %  %  %  %  %  %  %
   37m  %   %  %  %  %  %  %  %  %
 1;37m  %   %  %  %  %  %  %  %  %
```

Connected to online multiplayer text-based game.



```
12631/12237 103 7783/6722 115 4520 100 9
40000 74317386  who

-----------------------------------[ PEACE
FUL ]---------------------------------

Turtle Hermit   329 Pal Baathran Faeinth
, Sun's Paladin \|/PHOENIX\|/

You can currently see 1 player.
Max number this reboot 3 players.

12631/12237 103 7783/6722 115 4520 100 9
40000 74317386 █
```

Appendix D: Source Code

```c
/*
 *  ECE 3710 Lab 6: ascii.h
 *
 *
 */

#ifndef __ASCII_H
#define __ASCII_H

#include <string.h>

void get_ascii( unsigned char *, unsigned char );

#endif

/* END OF FILE */
//code_to_ascii.h

#ifndef CODE_TO_ASCII
#define CODE_TO_ASCII

void scan_code_init(void);

#endif
// dac.h

void DAC_init(void);
void DAC_beep(void);
//
//  lcd.h
//  ECE 3710 Microcontroller H&S
//  Utah State University
//

#ifndef __LCD_H
#define __LCD_H

#include "stm32f10x.h"

#define DISP_ORIENTATION 90

#if  ( DISP_ORIENTATION == 90 ) || ( DISP_ORIENTATION == 270 )

#define  MAX_X  320
#define  MAX_Y  240
//#define  CHARS_HORIZ_ON_Y 30
#define  CHARS_HORIZ_ON_Y 28
#define  CHARS_VERT_ON_Y 20
#define  CHARS_HORIZ_ON_X 40
#define  CHARS_VERT_ON_X 15

#elif  ( DISP_ORIENTATION == 0 ) || ( DISP_ORIENTATION == 180 )

#define  MAX_X  240
#define  MAX_Y  320
#define  CHARS_HORIZ_ON_Y 40
#define  CHARS_VERT_ON_Y 15

#endif


/* some LCD colors */
#define White          0xFFFF
#define Black          0x0000
#define Grey           0xF7DE
#define Blue           0x001F
#define Blue2          0x051F
#define Red            0xF800
#define Magenta        0xF81F
#define Green          0x07E0
```

```c
#define Cyan           0x7FFF
#define Yellow         0xFFE0


#define TermBlack Black
#define TermBlackBright 0x52aa
#define TermRed 0xa800
#define TermRedBright 0xfaaa
#define TermGreen 0x540
#define TermGreenBright 0x57ea
#define TermBrown 0xaaa0
#define TermBrownBright 0xffea
#define TermBlue 0x15
#define TermBlueBright 0x52bf
#define TermMagenta 0xa815
#define TermMagentaBright 0xfabf
#define TermCyan 0x555
#define TermCyanBright 0x57ff
#define TermWhite 0xa554
#define TermWhiteBright White
#define TermDefault TermWhite
#define TermDefaultBright TermWhiteBright


void LCD_Config(void);
void LCD_Initialization(void);
void LCD_Clear( unsigned short Color );

void LCD_WriteIndex( unsigned short index );
void LCD_WriteData( unsigned short data );
void LCD_Write_Generic(unsigned short toWrite, unsigned short dataBool);
void LCD_WriteReg( unsigned short LCD_Reg, unsigned short LCD_RegValue );

void LCD_SetCursor( unsigned short x, unsigned int y );
void delay_ms( unsigned int ms );

void LCD_DrawSquare( unsigned short x, unsigned short y, unsigned short h, unsigned short w, unsigned short color );


void LCD_DrawCharacterOnY (unsigned short x, unsigned short y, unsigned short fgColor, unsigned short bgColor, unsigned char symbol);
void LCD_WriteCharactersOnY (unsigned short x, unsigned short y, unsigned short fgColor, unsigned short bgColor, char* words, int maxLength);
void LCD_WriteLinesOnY(unsigned short x, unsigned short fgColor, unsigned short bgColor, char* words, char drawToLineEnd);

void LCD_DrawChar_rc (unsigned int row, unsigned int col, unsigned short fgColor, unsigned short bgColor, unsigned char symbol, unsigned char underline);

#endif

//  END OF FILE
//ps2_over_gpioc.h
#ifndef PS2_OVER_GPIOC
#define PS2_OVER_GPIOC

#define PS2_DATA_SIZE 1000



void ps2_over_gpioc_init(void);
//void ps2_dump_data_over_usart2(void);
int ps2_memcpy(unsigned char * dst);
void ps2_insert_to_buffer(char *insert, int size);


#endif


```

```
// terminal.h
#ifndef __TERMINAL
#define __TERMINAL

#include "lcd.h"

#define ROWS CHARS_VERT_ON_X // Num lines
#define COLS CHARS_HORIZ_ON_X // Num columns



void flushScreen(void);
void bufClear(void);
void handleAscii(unsigned char *buf, int bytes);

#endif

// usart2.h

#ifndef __USART2
#define __USART2
#include "stm32f10x.h"


#define USART2_DATA_SIZE 1000

void usart2_init(void);

void usart2_tx(unsigned char byte);
int usart2_memcpy(unsigned char * dst);

#endif
/*
 *  ECE 3710 Lab 6: ascii.c
 *
 */

#include "ascii.h"

static const unsigned char ascii[95][16] =
   {{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},  /* " " */
    {0x00,0x00,0x00,0x18,0x3C,0x3C,0x3C,0x18,0x18,0x00,0x18,0x18,0x00,0x00,0x00,0x00},  /* "!" */
    {0x00,0x00,0x00,0x66,0x66,0x66,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},  /* """ */
    {0x00,0x00,0x00,0x36,0x36,0x7F,0x36,0x36,0x36,0x7F,0x36,0x36,0x00,0x00,0x00,0x00},  /* "#" */
    {0x00,0x18,0x18,0x3C,0x66,0x60,0x30,0x18,0x0C,0x06,0x66,0x3C,0x18,0x18,0x00,0x00},  /* "$" */
    {0x00,0x00,0x70,0xD8,0xDA,0x76,0x0C,0x18,0x30,0x6E,0x5B,0x1B,0x0E,0x00,0x00,0x00},  /* "%" */
    {0x00,0x00,0x00,0x38,0x6C,0x6C,0x38,0x60,0x6F,0x66,0x66,0x3B,0x00,0x00,0x00,0x00},  /* "&" */
    {0x00,0x00,0x00,0x18,0x18,0x18,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},  /* "'" */
    {0x00,0x00,0x00,0x0C,0x18,0x18,0x30,0x30,0x30,0x30,0x30,0x18,0x18,0x0C,0x00,0x00},  /* "(" */
    {0x00,0x00,0x00,0x30,0x18,0x18,0x0C,0x0C,0x0C,0x0C,0x0C,0x18,0x18,0x30,0x00,0x00},  /* ")" */
    {0x00,0x00,0x00,0x00,0x00,0x36,0x1C,0x7F,0x1C,0x36,0x00,0x00,0x00,0x00,0x00,0x00},  /* "*" */
    {0x00,0x00,0x00,0x00,0x00,0x18,0x18,0x7E,0x18,0x18,0x00,0x00,0x00,0x00,0x00,0x00},  /* "+" */
    {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x1C,0x1C,0x0C,0x18,0x00,0x00},  /* "," */
    {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x7E,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},  /* "-" */
    {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x1C,0x1C,0x00,0x00,0x00,0x00},  /* "." */
    {0x00,0x00,0x00,0x06,0x06,0x0C,0x0C,0x18,0x18,0x30,0x30,0x60,0x60,0x00,0x00,0x00},  /* "/" */
    {0x00,0x00,0x00,0x1E,0x33,0x37,0x37,0x33,0x3B,0x3B,0x33,0x1E,0x00,0x00,0x00,0x00},  /* "0" */
    {0x00,0x00,0x00,0x0C,0x1C,0x7C,0x0C,0x0C,0x0C,0x0C,0x0C,0x0C,0x00,0x00,0x00,0x00},  /* "1" */
    {0x00,0x00,0x00,0x3C,0x66,0x66,0x06,0x0C,0x18,0x30,0x60,0x7E,0x00,0x00,0x00,0x00},  /* "2" */
    {0x00,0x00,0x00,0x3C,0x66,0x66,0x06,0x1C,0x06,0x66,0x66,0x3C,0x00,0x00,0x00,0x00},  /* "3" */
    {0x00,0x00,0x00,0x30,0x30,0x36,0x36,0x36,0x66,0x7F,0x06,0x06,0x00,0x00,0x00,0x00},  /* "4" */
    {0x00,0x00,0x00,0x7E,0x60,0x60,0x60,0x7C,0x06,0x06,0x0C,0x78,0x00,0x00,0x00,0x00},  /* "5" */
    {0x00,0x00,0x00,0x1C,0x18,0x30,0x7C,0x66,0x66,0x66,0x66,0x3C,0x00,0x00,0x00,0x00},  /* "6" */
    {0x00,0x00,0x00,0x7E,0x06,0x0C,0x0C,0x18,0x18,0x30,0x30,0x30,0x00,0x00,0x00,0x00},  /* "7" */
    {0x00,0x00,0x00,0x3C,0x66,0x66,0x76,0x3C,0x6E,0x66,0x66,0x3C,0x00,0x00,0x00,0x00},  /* "8" */
    {0x00,0x00,0x00,0x3C,0x66,0x66,0x66,0x66,0x3E,0x0C,0x18,0x38,0x00,0x00,0x00,0x00},  /* "9" */
    {0x00,0x00,0x00,0x00,0x00,0x1C,0x1C,0x00,0x00,0x00,0x1C,0x1C,0x00,0x00,0x00,0x00},  /* ":" */
    {0x00,0x00,0x00,0x00,0x00,0x1C,0x1C,0x00,0x00,0x00,0x1C,0x1C,0x0C,0x18,0x00,0x00},  /* ";" */
    {0x00,0x00,0x00,0x06,0x0C,0x18,0x30,0x60,0x30,0x18,0x0C,0x06,0x00,0x00,0x00,0x00},  /* "<" */
    {0x00,0x00,0x00,0x00,0x00,0x00,0x7E,0x00,0x7E,0x00,0x00,0x00,0x00,0x00,0x00,0x00},  /* "=" */
    {0x00,0x00,0x00,0x60,0x30,0x18,0x0C,0x06,0x0C,0x18,0x30,0x60,0x00,0x00,0x00,0x00},  /* ">" */
    {0x00,0x00,0x00,0x3C,0x66,0x66,0x0C,0x18,0x18,0x00,0x18,0x18,0x00,0x00,0x00,0x00},  /* "?" */
    {0x00,0x00,0x00,0x7E,0xC3,0xC3,0xCF,0xDB,0xDB,0xCF,0xC0,0x7F,0x00,0x00,0x00,0x00},  /* "@" */
    {0x00,0x00,0x00,0x18,0x3C,0x66,0x66,0x66,0x7E,0x66,0x66,0x66,0x00,0x00,0x00,0x00},  /* "A" */
    {0x00,0x00,0x00,0x7C,0x66,0x66,0x66,0x7C,0x66,0x66,0x66,0x7C,0x00,0x00,0x00,0x00},  /* "B" */
    {0x00,0x00,0x00,0x3C,0x66,0x66,0x60,0x60,0x60,0x66,0x66,0x3C,0x00,0x00,0x00,0x00},  /* "C" */
    {0x00,0x00,0x00,0x78,0x6C,0x66,0x66,0x66,0x66,0x66,0x6C,0x78,0x00,0x00,0x00,0x00},  /* "D" */
    {0x00,0x00,0x00,0x7E,0x60,0x60,0x60,0x7C,0x60,0x60,0x60,0x7E,0x00,0x00,0x00,0x00},  /* "E" */
    {0x00,0x00,0x00,0x7E,0x60,0x60,0x60,0x7C,0x60,0x60,0x60,0x60,0x00,0x00,0x00,0x00},  /* "F" */
    {0x00,0x00,0x00,0x3C,0x66,0x66,0x60,0x60,0x6E,0x66,0x66,0x3E,0x00,0x00,0x00,0x00},  /* "G" */
    {0x00,0x00,0x00,0x66,0x66,0x66,0x66,0x7E,0x66,0x66,0x66,0x66,0x00,0x00,0x00,0x00},  /* "H" */
    {0x00,0x00,0x00,0x3C,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x3C,0x00,0x00,0x00,0x00},  /* "I" */
    {0x00,0x00,0x00,0x06,0x06,0x06,0x06,0x06,0x06,0x66,0x66,0x3C,0x00,0x00,0x00,0x00},  /* "J" */
    {0x00,0x00,0x00,0x66,0x66,0x6C,0x6C,0x78,0x6C,0x6C,0x66,0x66,0x00,0x00,0x00,0x00},  /* "K" */
    {0x00,0x00,0x00,0x60,0x60,0x60,0x60,0x60,0x60,0x60,0x60,0x7E,0x00,0x00,0x00,0x00},  /* "L" */
    {0x00,0x00,0x00,0x63,0x63,0x77,0x6B,0x6B,0x6B,0x63,0x63,0x63,0x00,0x00,0x00,0x00},  /* "M" */
    {0x00,0x00,0x00,0x63,0x63,0x73,0x7B,0x6F,0x67,0x63,0x63,0x63,0x00,0x00,0x00,0x00},  /* "N" */
    {0x00,0x00,0x00,0x3C,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x3C,0x00,0x00,0x00,0x00},  /* "O" */
    {0x00,0x00,0x00,0x7C,0x66,0x66,0x66,0x7C,0x60,0x60,0x60,0x60,0x00,0x00,0x00,0x00},  /* "P" */
    {0x00,0x00,0x00,0x3C,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x3C,0x0C,0x06,0x00,0x00},  /* "Q" */
    {0x00,0x00,0x00,0x7C,0x66,0x66,0x66,0x7C,0x6C,0x66,0x66,0x66,0x00,0x00,0x00,0x00},  /* "R" */
    {0x00,0x00,0x00,0x3C,0x66,0x60,0x30,0x18,0x0C,0x06,0x66,0x3C,0x00,0x00,0x00,0x00},  /* "S" */
    {0x00,0x00,0x00,0x7E,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x00,0x00,0x00,0x00},  /* "T" */
    {0x00,0x00,0x00,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x3C,0x00,0x00,0x00,0x00},  /* "U" */
    {0x00,0x00,0x00,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x3C,0x18,0x00,0x00,0x00,0x00},  /* "V" */
    {0x00,0x00,0x00,0x63,0x63,0x63,0x6B,0x6B,0x6B,0x36,0x36,0x36,0x00,0x00,0x00,0x00},  /* "W" */
    {0x00,0x00,0x00,0x66,0x66,0x34,0x18,0x18,0x2C,0x66,0x66,0x66,0x00,0x00,0x00,0x00},  /* "X" */
    {0x00,0x00,0x00,0x66,0x66,0x66,0x66,0x3C,0x18,0x18,0x18,0x18,0x00,0x00,0x00,0x00},  /* "Y" */
    {0x00,0x00,0x00,0x7E,0x06,0x06,0x0C,0x18,0x30,0x60,0x60,0x7E,0x00,0x00,0x00,0x00},  /* "Z" */
    {0x00,0x00,0x00,0x3C,0x30,0x30,0x30,0x30,0x30,0x30,0x30,0x30,0x30,0x30,0x3C,0x00},  /* "[" */
    {0x00,0x00,0x00,0x60,0x60,0x30,0x30,0x18,0x18,0x0C,0x0C,0x06,0x06,0x00,0x00,0x00},  /* "\" */
    {0x00,0x00,0x00,0x3C,0x0C,0x0C,0x0C,0x0C,0x0C,0x0C,0x0C,0x0C,0x0C,0x0C,0x3C,0x00},  /* "]" */
    {0x00,0x18,0x3C,0x66,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},  /* "^" */
    {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xFF,0x00},  /* "_" */
    {0x00,0x00,0x00,0x18,0x18,0x18,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},  /* "`" */
    {0x00,0x00,0x00,0x00,0x00,0x3C,0x06,0x06,0x3E,0x66,0x66,0x3E,0x00,0x00,0x00,0x00},  /* "a" */
    {0x00,0x00,0x00,0x60,0x60,0x7C,0x66,0x66,0x66,0x66,0x66,0x7C,0x00,0x00,0x00,0x00},  /* "b" */
    {0x00,0x00,0x00,0x00,0x00,0x3C,0x66,0x60,0x60,0x60,0x66,0x3C,0x00,0x00,0x00,0x00},  /* "c" */
    {0x00,0x00,0x00,0x06,0x06,0x3E,0x66,0x66,0x66,0x66,0x66,0x3E,0x00,0x00,0x00,0x00},  /* "d" */
    {0x00,0x00,0x00,0x00,0x00,0x3C,0x66,0x66,0x7E,0x60,0x60,0x3C,0x00,0x00,0x00,0x00},  /* "e" */
    {0x00,0x00,0x00,0x1E,0x30,0x30,0x30,0x7E,0x30,0x30,0x30,0x30,0x00,0x00,0x00,0x00},  /* "f" */
    {0x00,0x00,0x00,0x00,0x00,0x3E,0x66,0x66,0x66,0x66,0x66,0x3E,0x06,0x06,0x7C,0x00},  /* "g" */
    {0x00,0x00,0x00,0x60,0x60,0x7C,0x66,0x66,0x66,0x66,0x66,0x66,0x00,0x00,0x00,0x00},  /* "h" */
    {0x00,0x00,0x18,0x18,0x00,0x78,0x18,0x18,0x18,0x18,0x18,0x7E,0x00,0x00,0x00,0x00},  /* "i" */
    {0x00,0x00,0x0C,0x0C,0x00,0x3C,0x0C,0x0C,0x0C,0x0C,0x0C,0x0C,0x0C,0x0C,0x78,0x00},  /* "j" */
    {0x00,0x00,0x00,0x60,0x60,0x66,0x66,0x6C,0x78,0x6C,0x66,0x66,0x00,0x00,0x00,0x00},  /* "k" */
    {0x00,0x00,0x00,0x78,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x7E,0x00,0x00,0x00,0x00},  /* "l" */
    {0x00,0x00,0x00,0x00,0x00,0x7E,0x6B,0x6B,0x6B,0x6B,0x6B,0x63,0x00,0x00,0x00,0x00},  /* "m" */
    {0x00,0x00,0x00,0x00,0x00,0x7C,0x66,0x66,0x66,0x66,0x66,0x66,0x00,0x00,0x00,0x00},  /* "n" */
    {0x00,0x00,0x00,0x00,0x00,0x3C,0x66,0x66,0x66,0x66,0x66,0x3C,0x00,0x00,0x00,0x00},  /* "o" */
    {0x00,0x00,0x00,0x00,0x00,0x7C,0x66,0x66,0x66,0x66,0x66,0x7C,0x60,0x60,0x60,0x00},  /* "p" */
    {0x00,0x00,0x00,0x00,0x00,0x3E,0x66,0x66,0x66,0x66,0x66,0x3E,0x06,0x06,0x06,0x00},  /* "q" */
    {0x00,0x00,0x00,0x00,0x00,0x66,0x6E,0x70,0x60,0x60,0x60,0x60,0x00,0x00,0x00,0x00},  /* "r" */
    {0x00,0x00,0x00,0x00,0x00,0x3E,0x60,0x60,0x3C,0x06,0x06,0x7C,0x00,0x00,0x00,0x00},  /* "s" */
    {0x00,0x00,0x00,0x30,0x30,0x7E,0x30,0x30,0x30,0x30,0x30,0x1E,0x00,0x00,0x00,0x00},  /* "t" */
    {0x00,0x00,0x00,0x00,0x00,0x66,0x66,0x66,0x66,0x66,0x66,0x3E,0x00,0x00,0x00,0x00},  /* "u" */
    {0x00,0x00,0x00,0x00,0x00,0x66,0x66,0x66,0x66,0x66,0x3C,0x18,0x00,0x00,0x00,0x00},  /* "v" */
    {0x00,0x00,0x00,0x00,0x00,0x63,0x6B,0x6B,0x6B,0x6B,0x36,0x36,0x00,0x00,0x00,0x00},  /* "w" */
    {0x00,0x00,0x00,0x00,0x00,0x66,0x66,0x3C,0x18,0x3C,0x66,0x66,0x00,0x00,0x00,0x00},  /* "x" */
    {0x00,0x00,0x00,0x00,0x00,0x66,0x66,0x66,0x66,0x66,0x66,0x3C,0x0C,0x18,0xF0,0x00},  /* "y" */
    {0x00,0x00,0x00,0x00,0x00,0x7E,0x06,0x0C,0x18,0x30,0x60,0x7E,0x00,0x00,0x00,0x00},  /* "z" */
    {0x00,0x00,0x00,0x0C,0x18,0x18,0x18,0x30,0x60,0x30,0x18,0x18,0x18,0x0C,0x00,0x00},  /* "{" */
    {0x00,0x00,0x00,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x00},  /* "|" */
    {0x00,0x00,0x00,0x30,0x18,0x18,0x18,0x0C,0x06,0x0C,0x18,0x18,0x18,0x30,0x00,0x00},  /* "}" */
    {0x00,0x00,0x00,0x71,0xDB,0x8E,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}}; /* "~" */

void get_ascii( unsigned char* buffer, unsigned char chr )
{
    memcpy( buffer, ascii[chr-32], 16 );
}
```

```c

/* END OF FILE */
//code_to_ascii.c

#include "code_to_ascii.h"

unsigned char map[0x100];
unsigned char shift_map[0x100];
unsigned char ctl_map[0x100];

void scan_code_init(void)
{
        int i = 0;
        for (i=0; i<0x100; i++)
        {
                map[i] = 0;
                shift_map[i] = 0;
                ctl_map[i] = 0;
        }

        map[0x1C]='a';
        map[0x32]='b';
        map[0x21]='c';
        map[0x23]='d';
        map[0x24]='e';
        map[0x2B]='f';
        map[0x34]='g';
        map[0x33]='h';
        map[0x43]='i';
        map[0x3B]='j';
        map[0x42]='k';
        map[0x4B]='l';
        map[0x3A]='m';
        map[0x31]='n';
        map[0x44]='o';
        map[0x4D]='p';
        map[0x15]='q';
        map[0x2D]='r';
        map[0x1B]='s';
        map[0x2C]='t';
        map[0x3C]='u';
        map[0x2A]='v';
        map[0x1D]='w';
        map[0x22]='x';
        map[0x35]='y';
        map[0x1A]='z';
        map[0x45]='0';
        map[0x16]='1';
        map[0x1E]='2';
        map[0x26]='3';
        map[0x25]='4';
        map[0x2E]='5';
        map[0x36]='6';
        map[0x3D]='7';
        map[0x3E]='8';
        map[0x46]='9';

        map[0x54]='[';
        map[0x0E]='`';
        map[0x4E]='-';
        map[0x55]='=';
        map[0x5D]='\\';
        map[0x52]='\'';
        map[0x49]='.';
        map[0x41]=',';
        map[0x4C]=';';
        map[0x5B]=']';
        map[0x4A]='/';

        map[0x29]=0x20;//SPACE
        map[0x5A]=0x0D;//ENTER
        map[0x76]=0x1B;//ESC
        map[0x66]=0x08;//BKSP
        map[0x0D]=0x09;//TAB

        map[0x7C]='*';//'KP *';
        map[0x79]='+';//'KP +';
        map[0x7B]='-';//'KP -';
        map[0x71]='.';//'KP .';
        map[0x70]='0';//'KP 0';
        map[0x69]='1';//'KP 1';
        map[0x72]='2';//'KP 2';
        map[0x7A]='3';//'KP 3';
        map[0x6B]='4';//'KP 4';
        map[0x73]='5';//'KP 5';
        map[0x74]='6';//'KP 6';
        map[0x6C]='7';//'KP 7';
        map[0x75]='8';//'KP 8';
        map[0x7D]='9';//'KP 9';


        shift_map[0x1C]='A';
        shift_map[0x32]='B';
        shift_map[0x21]='C';
        shift_map[0x23]='D';
        shift_map[0x24]='E';
        shift_map[0x2B]='F';
        shift_map[0x34]='G';
        shift_map[0x33]='H';
        shift_map[0x43]='I';
        shift_map[0x3B]='J';
        shift_map[0x42]='K';
        shift_map[0x4B]='L';
        shift_map[0x3A]='M';
        shift_map[0x31]='N';
        shift_map[0x44]='O';
        shift_map[0x4D]='P';
        shift_map[0x15]='Q';
        shift_map[0x2D]='R';
        shift_map[0x1B]='S';
        shift_map[0x2C]='T';
        shift_map[0x3C]='U';
        shift_map[0x2A]='V';
        shift_map[0x1D]='W';
        shift_map[0x22]='X';
        shift_map[0x35]='Y';
        shift_map[0x1A]='Z';
        shift_map[0x45]=')';
        shift_map[0x16]='!';
        shift_map[0x1E]='@';
        shift_map[0x26]='#';
        shift_map[0x25]='$';
        shift_map[0x2E]='%';
        shift_map[0x36]='^';
        shift_map[0x3D]='&';
        shift_map[0x3E]='*';
        shift_map[0x46]='(';

        shift_map[0x54]='{';
        shift_map[0x0E]='~';
        shift_map[0x4E]='_';
        shift_map[0x55]='+';
        shift_map[0x5D]='|';
        shift_map[0x52]='"';
        shift_map[0x49]='>';
        shift_map[0x41]='<';
        shift_map[0x4C]=':';
        shift_map[0x5B]='}';
        shift_map[0x4A]='?';

        shift_map[0x29]=0x20;//SPACE
        shift_map[0x5A]='\n';//ENTER - set to newline for debugging purposes
        shift_map[0x76]=0x1B;//ESC
        shift_map[0x66]=0x08;//BKSP
        shift_map[0x0D]=0x09;//TAB

        shift_map[0x7C]='*';//'KP *';
        shift_map[0x79]='+';//'KP +';
        shift_map[0x7B]='-';//'KP -';
        shift_map[0x71]='.';//'KP .';
        shift_map[0x70]='0';//'KP 0';
        shift_map[0x69]='1';//'KP 1';
        shift_map[0x72]='2';//'KP 2';
        shift_map[0x7A]='3';//'KP 3';
        shift_map[0x6B]='4';//'KP 4';
        shift_map[0x73]='5';//'KP 5';
        shift_map[0x74]='6';//'KP 6';
        shift_map[0x6C]='7';//'KP 7';
        shift_map[0x75]='8';//'KP 8';
        shift_map[0x7D]='9';//'KP 9';

        ctl_map[0x1e] = 0x00; //^@
        ctl_map[0x36] = 0x1e; //^^
        ctl_map[0x3e] = 0x7f; //^?
        ctl_map[0x4e] = 0x1f; //^_
        ctl_map[0x15] = 0x11; //^Q
        ctl_map[0x1d] = 0x17; //^W
        ctl_map[0x24] = 0x05; //^E
        ctl_map[0x2d] = 0x12; //^R
        ctl_map[0x2c] = 0x14; //^T
        ctl_map[0x35] = 0x19; //^Y
        ctl_map[0x3c] = 0x15; //^U
        ctl_map[0x44] = 0x0f; //^O
        ctl_map[0x4d] = 0x10; //^P
        ctl_map[0x54] = 0x1b; //^[
        ctl_map[0x5b] = 0x1d; //^]
        ctl_map[0x5d] = 0x1c; //^\\ ...
        ctl_map[0x1c] = 0x01; //^A
        ctl_map[0x1b] = 0x13; //^S
        ctl_map[0x2b] = 0x06; //^F
        ctl_map[0x34] = 0x07; //^G
        ctl_map[0x33] = 0x08; //^H
        ctl_map[0x3b] = 0x0a; //^J
        ctl_map[0x42] = 0x0b; //^K
        ctl_map[0x4b] = 0x0c; //^L
        ctl_map[0x1a] = 0x1a; //^Z
        ctl_map[0x22] = 0x18; //^X
        ctl_map[0x21] = 0x03; //^C
        ctl_map[0x2a] = 0x16; //^V
        ctl_map[0x32] = 0x02; //^B
        ctl_map[0x31] = 0x0e; //^N
        ctl_map[0x3a] = 0x0d; //^M
        ctl_map[0x23] = 0x04; //^D
        ctl_map[0x43] = 0x09; //^I

}
// dac.c

#include "stm32f10x.h"
#include "dac.h"


static int wave[40] = {
        2047,2367,2679,2976,3250,3494,3703,3870,3993,4068,4094,4068,3993,3870,3703,3494,3250,2976,
2679,2367,2047,1726,1414,1117,843,599,390,223,100,25,0,25,100,223,390,599,843,1117,1414,1726
};
int beepCount = 0;


void DAC_beep(void)
```

```c
{
        beepCount = 100;
}


void Tim3_init()
{
        RCC->APB1ENR |= RCC_APB1ENR_TIM3EN;
        NVIC->ISER[0] = NVIC_ISER_SETENA_29;
        NVIC->IP[7] = 0; // Highest priority!
        TIM3->CR1 = 0x94; // Count down, restart automatically, only update on under/overflow
        TIM3->DIER = 1; // enable interrupt on timer finish

        TIM3->ARR = 0xFFFF; // 8000 is 1ms on 8Mhz clock

        TIM3->CR1 |= 1; // enable timer.
}

void TIM3_IRQHandler()
// Interrupt on ISER[0]0x20000000
// Output new DAC value along the wave form
{
        if (beepCount)
        {
                static int count = 0;
                DAC->DHR12R2 = wave[count++];
                //DAC->SWTRIGR = 2;
                if (count == 40)
                        count = 0;
                beepCount--;
        }


        // reset interrupt pending in NVIC
        TIM3->SR &= 0xFFFFFFFE;
        NVIC->ICPR[0] = NVIC_ICPR_CLRPEND_29;
}

void DAC_init()
{
        // Enable gpio clock
        RCC->APB1ENR |= RCC_APB1ENR_DACEN;
        // Setup GPIOS
        // DAC_OUT2 is PA5
        RCC->APB2ENR |= RCC_APB2ENR_IOPAEN;
        GPIOA->CRL = (GPIOA->CRL & 0xFF0FFFFF) | 0x00B00000; // PC5 set to output AF push-pull

        // Write configs
        DAC->CR = 0x010000; // enable DAC channel 2, turn off buffering
        //DAC->CR |= 0x3C0000; // enable triggers on software trigger

        Tim3_init();
}


//
//  lcd.c
//  ECE 3710 Microcontroller H&S
//  Utah State University
//  Written by Kelly Hathaway
//  And William Hatch and Scott Sorensen
//

#include "stm32f10x.h"
#include "lcd.h"
#include "ascii.h"

#define WR_low_Pin  0x0002  // Pin 1
#define RD_low_Pin  0x0004  // Pin 2
#define CS_low_Pin  0x0040  // Pin 6
```

```
#define DC_Pin      0x0080  // Pin 7


//  configuration of the LCD port pins
void LCD_Config(void)
{
    unsigned int config_temp;

    RCC->APB2ENR |= 0x1D;       // Enable port A, B, and C

    config_temp  = GPIOA->CRL;  // Pin A.3 for Back light
    config_temp &= ~0x0000F000;
    config_temp |=  0x00003000;
    GPIOA->CRL   = config_temp;

    GPIOB->CRL   = 0x33333333;  // Port B for Data[15:0] pins
    GPIOB->CRH   = 0x33333333;

    config_temp  = GPIOC->CRL;  // PC.0(LCD RST), PC.1(WR), PC.2(RD) , PC.6(CS), PC.7(DC)
    config_temp &= ~0xFF000FFF;
    config_temp |=  0x33000333;
    GPIOC->CRL   = config_temp;
}

void LCD_Initialization(void)
{
    unsigned int config_temp;

    LCD_Config();

    config_temp  = AFIO->MAPR;  // enable SW Disable JTAG
    config_temp &= ~0x07000000;
    config_temp |=  0x02000000;
    AFIO->MAPR   = config_temp;

    GPIOC->BRR  = 0x0001;   // LCD reset
    delay_ms(100);
    GPIOC->BSRR = 0x0001;
    GPIOA->BSRR = 0x0008;   // back light

    LCD_WriteReg(0x0000,0x0001);    delay_ms(50);   /* Enable LCD Oscillator */
    LCD_WriteReg(0x0003,0xA8A4);    delay_ms(50);   // Power control(1)
    LCD_WriteReg(0x000C,0x0000);    delay_ms(50);   // Power control(2)
    LCD_WriteReg(0x000D,0x080C);    delay_ms(50);   // Power control(3)
    LCD_WriteReg(0x000E,0x2B00);    delay_ms(50);   // Power control(4)
    LCD_WriteReg(0x001E,0x00B0);    delay_ms(50);   // Power control(5)
    LCD_WriteReg(0x0001,0x2B3F);    delay_ms(50);   // Driver Output Control /* 320*240 0x2B3F */
    LCD_WriteReg(0x0002,0x0600);    delay_ms(50);   // LCD Drive AC Control
    LCD_WriteReg(0x0010,0x0000);    delay_ms(50);   // Sleep Mode off
   // LCD_WriteReg(0x0011,0x6070);    delay_ms(50);   // Entry Mode                    ## flip bit 3 to switch horiz/vert auto-update on write
                LCD_WriteReg(0x0011,0x6078);    delay_ms(50);   // Entry Mode ## flip bit 3 to switch horiz/vert auto-update on write
    LCD_WriteReg(0x0005,0x0000);    delay_ms(50);   // Compare register(1)
    LCD_WriteReg(0x0006,0x0000);    delay_ms(50);   // Compare register(2)
    LCD_WriteReg(0x0016,0xEF1C);    delay_ms(50);   // Horizontal Porch
    LCD_WriteReg(0x0017,0x0003);    delay_ms(50);   // Vertical Porch
    LCD_WriteReg(0x0007,0x0133);    delay_ms(50);   // Display Control
    LCD_WriteReg(0x000B,0x0000);    delay_ms(50);   // Frame Cycle control
    LCD_WriteReg(0x000F,0x0000);    delay_ms(50);   // Gate scan start position
    LCD_WriteReg(0x0041,0x0000);    delay_ms(50);   // Vertical scroll control(1)
    LCD_WriteReg(0x0042,0x0000);    delay_ms(50);   // Vertical scroll control(2)
    LCD_WriteReg(0x0048,0x0000);    delay_ms(50);   // First window start
    LCD_WriteReg(0x0049,0x013F);    delay_ms(50);   // First window end
    LCD_WriteReg(0x004A,0x0000);    delay_ms(50);   // Second window start
    LCD_WriteReg(0x004B,0x0000);    delay_ms(50);   // Second window end
    LCD_WriteReg(0x0044,0xEF00);    delay_ms(50);   // Horizontal RAM address position
    LCD_WriteReg(0x0045,0x0000);    delay_ms(50);   // Vertical RAM address start position
    LCD_WriteReg(0x0046,0x013F);    delay_ms(50);   // Vertical RAM address end position
    LCD_WriteReg(0x0030,0x0707);    delay_ms(50);   // gamma control(1)
    LCD_WriteReg(0x0031,0x0204);    delay_ms(50);   // gamma control(2)
    LCD_WriteReg(0x0032,0x0204);    delay_ms(50);   // gamma control(3)
    LCD_WriteReg(0x0033,0x0502);    delay_ms(50);   // gamma control(4)
    LCD_WriteReg(0x0034,0x0507);    delay_ms(50);   // gamma control(5)
    LCD_WriteReg(0x0035,0x0204);    delay_ms(50);   // gamma control(6)
    LCD_WriteReg(0x0036,0x0204);    delay_ms(50);   // gamma control(7)
    LCD_WriteReg(0x0037,0x0502);    delay_ms(50);   // gamma control(8)
    LCD_WriteReg(0x003A,0x0302);    delay_ms(50);   // gamma control(9)
    LCD_WriteReg(0x003B,0x0302);    delay_ms(50);   // gamma control(10)
    LCD_WriteReg(0x0023,0x0000);    delay_ms(50);   // RAM write data mask(1)
    LCD_WriteReg(0x0024,0x0000);    delay_ms(50);   // RAM write data mask(2)
    LCD_WriteReg(0x0025,0x8000);    delay_ms(50);   // Frame Frequency
    LCD_WriteReg(0x004f,0);                         // Set GDDRAM Y address counter
    LCD_WriteReg(0x004e,0);                         // Set GDDRAM X address counter

    delay_ms(50);
}

// Paints the LCD with Color
void LCD_Clear( unsigned short Color )
{
    unsigned int i;

    LCD_SetCursor(0,0);

    GPIOC->BRR  = CS_low_Pin;

    LCD_WriteIndex( 0x0022 );
    for( i=0; i< MAX_X*MAX_Y; i++ )
        LCD_WriteData( Color );

    GPIOC->BSRR = CS_low_Pin;
}

// Write a command
void LCD_WriteIndex( unsigned short index )
{
  LCD_Write_Generic(index, 0);
}

// Write data
void LCD_WriteData( unsigned short data )
{
  LCD_Write_Generic(data, 1);
}

// Write generic...
void LCD_Write_Generic(unsigned short toWrite, unsigned short dataBool)
{
        unsigned short pc_ops = GPIOC->ODR;

        // Configure Ports - done in LCD init function
        // Set control bits (RD, WR, D/C, CS)
        // PC.0(LCD RST = ?), PC.1(WR = 1, then 0, then 1), PC.2(RD = 1) , PC.6(CS = unset then set), PC.7(DC = dataBool)

        pc_ops &= 0xFF38; // unset RST,WR,RD,CS,DC
        pc_ops |= 0x0007; // set RD = 1, and WR = 1, and RST = 1
        if (dataBool) pc_ops |= 0x80; // set DC = 1 if we want to write data

        GPIOC->ODR = pc_ops;
        delay_ms(0);

        // Write data bits
        GPIOB->ODR = toWrite;
        delay_ms(0);

        GPIOC->ODR = (pc_ops & 0xFFFD); // unset WR
        GPIOC->ODR = pc_ops; // set WR

```

```
}

//
void LCD_WriteReg( unsigned short LCD_Reg, unsigned short LCD_RegValue )
{
    GPIOC->BRR  = CS_low_Pin;

    LCD_WriteIndex( LCD_Reg );
    LCD_WriteData( LCD_RegValue );

    GPIOC->BSRR = CS_low_Pin;
}

// Set cursor to x y address
void LCD_SetCursor( unsigned short x, unsigned int y )
{
    #if   ( DISP_ORIENTATION == 90 ) || ( DISP_ORIENTATION == 270 )

        unsigned short swap_temp;

        y = (MAX_Y-1) - y;
        swap_temp = y;
        y = x;
        x = swap_temp;

    #elif ( DISP_ORIENTATION ==  0 ) || ( DISP_ORIENTATION == 180 )

        y = (MAX_Y-1) - y;

    #endif

    LCD_WriteReg( 0x004E, x );
    LCD_WriteReg( 0x004F, y );
}

void delay_ms( unsigned int ms )
{
        int i;
        while(ms--)
        {
                //for(i = 0; i < 1669; ++i); // 1 ms delay loop
                for(i = 0; i < 8676; ++i); // 1 ms delay loop
        }
}


void LCD_DrawSquareY( unsigned short x, unsigned short y, unsigned short w, unsigned short h, unsigned short color )
{
        unsigned int i,j;

    LCD_SetCursor(x,y);

    GPIOC->BRR  = CS_low_Pin;


                for (j=0; j < w; j++ )
          {
                        LCD_SetCursor(x+j, y);
                        LCD_WriteIndex( 0x0022 );
                        for( i=0; i < h; i++ )
                                        LCD_WriteData( color );
    }

    GPIOC->BSRR = CS_low_Pin;
}
void LCD_DrawSquare( unsigned short x, unsigned short y, unsigned short w, unsigned short h, unsigned short color )
{
        unsigned int i,j;

    LCD_SetCursor(x,y);

    GPIOC->BRR  = CS_low_Pin;


                for (j=0; j < w; j++ )
          {
                        LCD_SetCursor(x, y+j);
                        LCD_WriteIndex( 0x0022 );
                        for( i=0; i < h; i++ )
                                        LCD_WriteData( color );
    }

    GPIOC->BSRR = CS_low_Pin;
}

void LCD_DrawCharacterOnY (unsigned short x, unsigned short y, unsigned short fgColor, unsigned short bgColor, unsigned char symbol)
// Draws a character oriented so that left to right goes along the positive y axis
{
        unsigned char ascii_buf[16];
        unsigned char line;
        int i, j;

        LCD_SetCursor(x,y);
        GPIOC->BRR = CS_low_Pin;

        get_ascii(ascii_buf, symbol);

        for (i = 0; i < 16; ++i)
        {
                line = ascii_buf[i];
                LCD_SetCursor(x+i, y);
                LCD_WriteIndex( 0x0022 );
                for (j = 0; j < 8; ++j)
                {
                        if (line & (0x80 >> j))
                                LCD_WriteData( fgColor );
                        else
                                LCD_WriteData( bgColor );
                        //delay_ms(1);
                }
        }
        GPIOC->BSRR = CS_low_Pin;
}
void LCD_DrawCharacterOnX (unsigned short x, unsigned short y, unsigned short fgColor, unsigned short bgColor, unsigned char symbol, unsigned char underline)
// Draws a character oriented so that left to right goes along the positive X axis
{
        unsigned char ascii_buf[16];
        unsigned char line;
        int i, j;

        LCD_SetCursor(x,y);
        GPIOC->BRR = CS_low_Pin;

        get_ascii(ascii_buf, symbol);

        for (i = 0; i < 16; ++i)
        {
                line = ascii_buf[i];
                if (i == 15 && underline)
                        line = 0xFF;
                LCD_SetCursor(x, y+i);
                LCD_WriteIndex( 0x0022 );
                for (j = 0; j < 8; ++j)
                {
                        if (line & (0x80 >> j))
                                LCD_WriteData( fgColor );
                        else
                                LCD_WriteData( bgColor );
                        //delay_ms(1);
                }
        }
        GPIOC->BSRR = CS_low_Pin;
}
void LCD_DrawChar_rc (unsigned int row, unsigned int col, unsigned short fgColor, unsigned short bgColor, unsigned char symbol, unsigned char underline)
// Draws a character on the givel row and column
{
        //LCD_DrawCharacterOnY(row * 16, MAX_Y - 12 - (col*8), fgColor, bgColor, symbol);
        LCD_DrawCharacterOnX(col*8, row*16, fgColor, bgColor, symbol, underline);
}

void LCD_WriteCharactersOnY (unsigned short x, unsigned short y, unsigned short fgColor, unsigned short bgColor, char* words, int maxLength)
// Draws a line of characters increasing on Y axis
{
        int i;
        for (i = 0; i < maxLength; ++i)
        {
                if(words[i] == 0)
                        break;
                LCD_DrawCharacterOnY(x, y - (8*i), fgColor, bgColor, words[i]);
        }
}
void LCD_WriteLinesOnY(unsigned short lineNumber, unsigned short fgColor, unsigned short bgColor, char* words, char drawToLineEnd)
{
        int len, numLines, lastLineLength, i, curLine;
        char spaces[CHARS_HORIZ_ON_Y];
        len = strlen(words);
        numLines = len / CHARS_HORIZ_ON_Y;
        lastLineLength = len % CHARS_HORIZ_ON_Y;
        if(lastLineLength != 0)
                numLines++;

        for(curLine = 0; curLine < numLines; ++curLine)
        {
                LCD_WriteCharactersOnY((lineNumber + curLine) * 16, MAX_Y-12, fgColor, bgColor, words+((CHARS_HORIZ_ON_Y) * curLine), CHARS_HORIZ_ON_Y);
        }
        // TODO: fix draw to end of last line
        if(lastLineLength != 0)
        {
                for(i = 0; i < CHARS_HORIZ_ON_Y; ++i)
                {
                        spaces[i] = ' ';
                }
                LCD_WriteCharactersOnY((lineNumber + curLine - 1) * 16, MAX_Y-12-(lastLineLength*8), fgColor, bgColor, spaces, CHARS_HORIZ_ON_Y-lastLineLength);
        }
}




//  END OF FILE
// main.c
// ECE 3710
// Final Project
// By William Hatch and Scott Sorensen


#include "stm32f10x.h"
#include "lcd.h"
#include "usart2.h"
#include "ps2_over_gpioc.h"
#include "code_to_ascii.h"
```

```c
#include "terminal.h"
#include "dac.h"



void SystemInit(void)
{
}

void handlePs2Data(void);
void handleUsartData(void);



extern int cursorCol, cursorRow;
extern char screenChars[ROWS][COLS];
extern unsigned short screenFgColor[ROWS][COLS];
extern unsigned short screenBgColor[ROWS][COLS];



int main()
{
        // Enable external oscillator
  //RCC->CFGR = 0x0418000A;                     // Mult PLL by 8 = 32 MHz
        RCC->CFGR = 0x0428000A;                     // 48MHz
        /// 418->428 makes it 48Mhz
  RCC->CR = 0x03004583; //USe PLL Clock for SW and MC


        // Initialize everything
        LCD_Initialization();
        usart2_init();
        ps2_over_gpioc_init();

        LCD_Clear(Black);
        scan_code_init();
        DAC_init();
        bufClear();

        while(1)
        {
                handlePs2Data();
                handleUsartData();
                flushScreen();
        }

}

void handlePs2Data()
{
        int bytes;
        int i;
        unsigned char buf[PS2_DATA_SIZE];

        // copy buffer
        bytes = ps2_memcpy(buf);

        // tx buffer
        for (i = 0; i < bytes; ++i)
        {
                usart2_tx(buf[i]);
        }
}

void handleUsartData()
{
        int bytes;
        unsigned char buf[USART2_DATA_SIZE];
        // copy buffer
        bytes = usart2_memcpy(buf);

        // push buffered data to display
        handleAscii(buf, bytes);
}


void HardFault_Handler()
{
        LCD_Clear(Red);
        //Reset_Handler();
}

//ps2_over_gpioc.c

#include "stm32f10x.h"
#include "ps2_over_gpioc.h"
#include "usart2.h"
#include "code_to_ascii.h"
#include "lcd.h"

unsigned char ps2_data[PS2_DATA_SIZE];
int ps2_bytes_rec = 0;

extern unsigned char map[0x100];
extern unsigned char shift_map[0x100];
extern unsigned char ctl_map[0x100];

void ps2_insert_to_buffer(char *insert, int size)
{
        int i;

        if (ps2_bytes_rec + size > PS2_DATA_SIZE)
                return;

        NVIC->ICER[0] = NVIC_ICER_CLRENA_10;
        for (i = 0; i < size; ++i)
        {
                ps2_data[ps2_bytes_rec++] = insert[i];
        }
        NVIC->ISER[0] = NVIC_ISER_SETENA_10;
}

// Switch PC0 to PC4, and PC1 to PC3
void ps2_over_gpioc_init(void)
{
        // Setup EXTI0
        EXTI->IMR |= EXTI_IMR_MR4;
        //EXTI->EMR |= EXTI_EMR_MR4;
        EXTI->RTSR |= EXTI_RTSR_TR4;

    RCC->APB2ENR |= RCC_APB2ENR_IOPCEN;

        // Setup AFIO events
        RCC->APB2ENR |= 1; // enable AFIO clock
        //AFIO->EVCR = 0b10100000; // Events enabled on PC0
        //AFIO->EVCR = AFIO_EVCR_EVOE | AFIO_EVCR_PORT_PC | AFIO_EVCR_PIN_PX0;
        AFIO->EXTICR[1] = (AFIO->EXTICR[2] & 0xFFFFFFF0) | 0x2;
        GPIOC->CRL = (GPIOC->CRL & 0xFFF00FFF) | 0x88000; // Configure PC[3-4] for pull-up/pull-down input
        GPIOC->ODR |= 0x18; // set  bits of ODR so we can read the input.

        // Setup NVIC
        // EXT4 is interrupt number 10
        NVIC->ISER[0] = NVIC_ISER_SETENA_10;

        // Set priority - this should be our high priority interrupt
        //NVIC->IP[1] = (NVIC->IP[1] & 0xFF00FFFF) | 0x00FF0000;

}
```

```
void EXTI4_IRQHandler(void)
{
        static unsigned int calls = 0;
        static unsigned char rx_state = 0; // 0 not receiving, 1 receiving
        static unsigned char data_bits_rec = 0;
        static unsigned char data = 0;
        static unsigned char stop_bit_rec = 0;
        static unsigned char last_byte_was_escape = 0;
        static unsigned char shift_on = 0;
        static unsigned char ctl_on = 0;
        static unsigned char alt_on = 0;

        unsigned char bit, ascii;

        calls++;
        bit = GPIOC->IDR & 0x8; // read pc3 (data line)
        bit = bit << 4;
        while(1)
  {
        // Handle start bit
        if (!rx_state)
        {
                if(bit)
                        break;
                rx_state = 1;
                data_bits_rec = 0;
                data = 0;
                stop_bit_rec = 0;

                break;
        }

        // Receive data
        if (data_bits_rec < 8)
        {
                data = data >> 1;
                data |= bit;
                data_bits_rec++;
                break;
        }

        // Handle stop/acknowledge bits
        if (! stop_bit_rec)
        {
                stop_bit_rec = 1;


                // Map scancodes to ascii, throwing away everything but lowercase
                // alphanumeric characters and spaces.

                if (last_byte_was_escape)
                {
                        last_byte_was_escape = 0;
                        if(shift_on && (data == 0x12 | data == 0x59))
                                shift_on = 0;
                        if(ctl_on && (data == 0x14))
                                ctl_on = 0;
                        if(alt_on && (data == 0x11))
                                alt_on = 0;
                        break;
                }
                if (data == 0xF0) // key up escape
                {
                        last_byte_was_escape = 1;
                        break;
                }

                if (data == 0x12 | data == 0x59) // key down shift
                {
                        shift_on = 1;
                        break;
                }
                if (data == 0x14) // key down ctl
                {
                        ctl_on = 1;
                        break;
                }
                if (data == 0x11) // key down alt
                {
                        alt_on = 1;
                        break;
                }

                if (ctl_on)
                        ascii = ctl_map[data];
                else if (shift_on)
                        ascii = shift_map[data];
                else
                        ascii = map[data];




                // put data into buffer
                if (ps2_bytes_rec < PS2_DATA_SIZE-1)
                {
                        if(alt_on)
                        {
                                ps2_data[ps2_bytes_rec] = 0x1b; // escape
                                ps2_bytes_rec++;
                        }
                        ps2_data[ps2_bytes_rec] = ascii;
                        //ps2_data[ps2_bytes_rec] = data;
                        ps2_bytes_rec++;

                }

                break;
        }


        // Here we will let one clock cycle pass, because there may
        // be an acknowledgement bit, but we're not sure how to handle it.

        rx_state = 0;
        break;
    }

          //NVIC->ICPR[0] = 0x40;
                EXTI->PR = 0x10; // Clear EXTI pending
                NVIC->ICPR[0] = NVIC_ICPR_CLRPEND_10;
    return;
}

void ps2_dump_data_over_usart2()
// Dumps data in ps2_data over usart2
{
        int i;

        for (i = 0; i < ps2_bytes_rec; ++i)
        {
                usart2_tx(ps2_data[i]);
                LCD_DrawCharacterOnY(40,40,Blue, Black, ps2_data[i]);
        }
        ps2_bytes_rec =0;
}

int ps2_memcpy(unsigned char * dst)
{
        int ret, i;
        // turn off interrupts
        NVIC->ICER[0] = NVIC_ICER_CLRENA_10;
        // copy buffer
        for(i = 0; i < ps2_bytes_rec; ++i)
        {
                dst[i] = ps2_data[i];
        }
        ret = ps2_bytes_rec;

        ps2_bytes_rec = 0;
        // enable interrupts
        NVIC->ISER[0] = NVIC_ISER_SETENA_10;
        return ret;
}



// terminal.c

#include "terminal.h"
#include "lcd.h"
#include "ps2_over_gpioc.h"
#include "dac.h"

#define CURSOR_STACK_SIZE 200
int cursorStack[2][2][CURSOR_STACK_SIZE]; // one for CSI stack, one for ESC (non-CSI stack)
int CSIcursorStackPointer=-1;
int ESCcursorStackPointer=-1;
int cursorCol = 0; // cursor column number
int cursorRow = 0; // cursor line number

unsigned short currFgColor = TermDefault;
unsigned short currBgColor = TermBlack;

unsigned char currDisplayOps = 0; // for flags for underline, blink, etc

char screenChars[ROWS][COLS];
char oldScreenChars[ROWS][COLS];
unsigned short screenFgColor[ROWS][COLS];
unsigned short screenBgColor[ROWS][COLS];
unsigned short oldScreenFgColor[ROWS][COLS];
unsigned short oldScreenBgColor[ROWS][COLS];
int screenTop = 0;
int oldScreenTop = 0;

unsigned char screenDisplayOps[ROWS][COLS];
unsigned char oldScreenDisplayOps[ROWS][COLS];
unsigned char drawCursor = 1;
#define CURSOR_CHAR 1 // attribute for character that the cursor is on
#define UNDERLINE 2
#define BOLD 4
#define BLINK 8
#define REVERSE_VIDEO 16
#define HALF_BRIGHT 32



//////////////////////// Ascii Control Functions //////////////////////

void addCursorAttr()
{ // add cursor attribute to screenChar at current cursor location
        int rowTr; // translated row
        rowTr = (cursorRow + screenTop) % ROWS;
        screenDisplayOps[rowTr][cursorCol] |= CURSOR_CHAR;
}
void remCursorAttr()
{ // remove cursor attribute to screenChar at current cursor location
        int rowTr; // translated row
        rowTr = (cursorRow + screenTop) % ROWS;
```

```c
        screenDisplayOps[rowTr][cursorCol] &= ~CURSOR_CHAR;
}
void clearExtraneousCursors()
{ // since we have extra cursors, let's add this cludge to clear them.
        int i,j;
        for(i = 0; i < ROWS; ++i)
        {
                for(j = 0; j < COLS; ++j)
                {
                        screenDisplayOps[i][j] &= ~CURSOR_CHAR;
                }
        }
        addCursorAttr();
}

void do_LF()
// Do a standard line feed
{
        int i, rowTr;

        remCursorAttr();
        if (cursorRow < ROWS -1)
                cursorRow = (cursorRow + 1);
        else
        {
                screenTop = (screenTop+1) % ROWS;
                rowTr = (ROWS - 1 + screenTop) % ROWS;
                for(i = 0; i < COLS; ++i)
                {
                        screenChars[rowTr][i] = ' ';
                }
        }
        addCursorAttr();
}

void do_CR()
// Do a standard carriage return
{
        remCursorAttr();
        cursorCol = 0;
        addCursorAttr();
}

void handle_LF()
// Handle the \n character
// In newline mode this gives both LF and CR
{
        do_LF();
        // TODO - check somehow whether newline mode is on (it probably always will be
        // if (newLineMode)
                do_CR();
}

void handle_CR()
// handle CR character
{
        // I think this may be the place to do this line blackout...
        int j;
        for(j = cursorCol; j < COLS; ++j)
        {
                screenChars[(cursorRow + screenTop)%ROWS][j] = ' ';
        }
        do_CR();
}

void advance_cursor()
// advances the cursor, duh.
{
        int i;
        remCursorAttr();
        if(cursorCol >= COLS - 1)
        {   // It seems many terminals don't automatically scroll, and it breaks some functionality.
                  // Unless I make a termcap entry for this terminal or figure out how it works, it breaks functionality of
                  // programs like vim in screen... odd...
                cursorCol = 0;
                if (cursorRow < ROWS -1)
                        cursorRow = (cursorRow +1);
                else
                { // Scroll screen
                        screenTop = (screenTop +1) % ROWS;
                        for (i = 0; i < COLS; ++i)
                        { // clear the new bottom line to be blank as it scrolls in.
                                screenChars[(screenTop+ROWS-1)%ROWS][i] = ' ';
                                screenDisplayOps[(screenTop+ROWS-1)%ROWS][i] = 0;
                                screenFgColor[(screenTop+ROWS-1)%ROWS][i] = TermDefault;
                                screenBgColor[(screenTop+ROWS-1)%ROWS][i] = TermBlack;
                        }
                }
        }
        else
                cursorCol++;
        addCursorAttr();
}

void handle_normal(char ascii)
// handles normal characters for printing to the screen
{
        int rowTr; // translated row
        rowTr = (cursorRow + screenTop) % ROWS;
        screenChars[rowTr][cursorCol] = ascii;
        screenFgColor[rowTr][cursorCol] = currFgColor;
        screenBgColor[rowTr][cursorCol] = currBgColor;
        screenDisplayOps[rowTr][cursorCol] = currDisplayOps;
        advance_cursor();
}

void moveCursor(unsigned int num, unsigned int dir)
{
        #define UP 1
        #define DOWN 2
        #define LEFT 3
        #define RIGHT 4
        remCursorAttr();
        if (dir == UP)
        {
                cursorRow -= num;
                if (cursorRow < 0)
                        cursorRow = 0;
        }
        else if (dir == DOWN)
        {
                cursorRow += num;
                if (cursorRow > ROWS -1)
                        cursorRow = ROWS - 1;
        }
        else if (dir == RIGHT)
        {
                cursorCol += num;
                if (cursorCol > COLS - 1)
                        cursorCol = COLS -1;
        }
        else if (dir == LEFT)
        {
                cursorCol -= num;
                if (cursorCol < 0)
                        cursorCol = 0;
        }
        addCursorAttr();
```

```c
}

void handle_colorCodes(unsigned int *csi_numbers, unsigned int csi_numbers_rec)
{
        int i;
        for (i = 0; i < csi_numbers_rec; ++i)
        {
                switch (csi_numbers[i])
                {
                        case 0:
                                currBgColor = TermBlack;
                                currFgColor = TermDefault;
                                currDisplayOps = 0;
                                break;
                        case 1:
                                currDisplayOps |= BOLD;
                                break;
                        case 2:
                                // set half-bright
                                currDisplayOps |= HALF_BRIGHT;
                                break;
                        case 4:
                                // set underscore
                                currDisplayOps |= UNDERLINE;
                                break;
                        case 5:
                                // set blink
                                currDisplayOps |= BLINK;
                                break;
                        case 7:
                                // set reverse video (whatever that is)
                                currDisplayOps |= REVERSE_VIDEO;
                                break;
                        case 21:
                                // set normal intensity (bold off?)
                                currDisplayOps &= ~BOLD;
                                break;
                        case 22:
                                // set normal intensity (half-bright off?)
                                currDisplayOps &= ~HALF_BRIGHT;
                                break;
                        case 24:
                                // set underline off
                                currDisplayOps &= ~UNDERLINE;
                                break;
                        case 25:
                                // set blink off
                                currDisplayOps &= ~BLINK;
                                break;
                        case 27:
                                // set reverse video off
                                currDisplayOps &= ~REVERSE_VIDEO;
                                break;

                        // 30-49 are mostly basic color options
                        case 30:
                                currFgColor = TermBlack;
                                if(currDisplayOps & BOLD)
                                        currFgColor = TermBlackBright;
                                break;
                        case 31:
                                currFgColor = TermRed;
                                if(currDisplayOps & BOLD)
                                        currFgColor = TermRedBright;
                                break;
                        case 32:
                                currFgColor = TermGreen;
                                if(currDisplayOps & BOLD)
                                        currFgColor = TermGreenBright;
                                break;
```

```
				case 33:
					currFgColor = TermBrown;
					if(currDisplayOps & BOLD)
							currFgColor = TermBrownBright;
					break;
				case 34:
					currFgColor = TermBlue;
					if(currDisplayOps & BOLD)
						currFgColor = TermBlueBright;
					break;
				case 35:
					currFgColor = TermMagenta;
					if(currDisplayOps & BOLD)
						currFgColor = TermMagentaBright;
					break;
				case 36:
					currFgColor = TermCyan;
					if(currDisplayOps & BOLD)
						currFgColor = TermCyanBright;
					break;
				case 37:
					currFgColor = TermWhite;
					if(currDisplayOps & BOLD)
						currFgColor = TermWhiteBright;
					break;
				case 38:
					currFgColor = TermDefault;
					currDisplayOps |= UNDERLINE;
					break;
				case 39:
					currFgColor = TermDefault;
					currDisplayOps &= ~UNDERLINE;
					break;
				case 40:
					currBgColor = TermBlack;
					break;
				case 41:
					currBgColor = TermRed;
					break;
				case 42:
					currBgColor = TermGreen;
					break;
				case 43:
					currBgColor = TermBrown;
					break;
				case 44:
					currBgColor = TermBlue;
					break;
				case 45:
					currBgColor = TermMagenta;
					break;
				case 46:
					currBgColor = TermCyan;
					break;
				case 47:
					currBgColor = TermWhite;
					break;
				case 48:
					// ???
					break;
				case 49:
					currBgColor = TermBlack;
					break;

				default:
					break;
			}
		}
	}

```

```
void keepCursorInBounds()
{
        if (cursorRow >= ROWS)
        {
                cursorRow = ROWS-1;
        }
        else if (cursorRow < 0)
                cursorRow = 0;
        if (cursorCol >= COLS-1)
                cursorCol = COLS-1;
        else if (cursorCol < 0)
                cursorCol = 0;
}

void reportCursorToHost()
{
        char rowColResponse[10];
        char rowNum[3];
        char colNum[3];
        int divisor, i,j;
        char firstSeen = 0;
        int rowsSent=0, colsSent=0;
        // echo ESC[<ROW>;<COL>R

        for (i = 0, divisor = 100; divisor; ++i, divisor/=10)
        {
                rowNum[i] = ((cursorRow+1)/divisor) % 10;
                colNum[i] = ((cursorCol+1)/divisor) % 10;
        }
        rowColResponse[0] = 033; // ESC
        rowColResponse[1]='[';
        for(i = 2, j = 0; j < 4; ++j) // i is response position, j is loop position
        {
                if(rowNum[j] || j == 3) // only start putting out characters if we've seen the first character or it's all 0
                        firstSeen = 1;
                if(firstSeen)
                {
                        rowColResponse[i] = rowNum[j];
                        ++i;
                        ++rowsSent;
                }
        }
        rowColResponse[i++] = ';';
        firstSeen = 0;
        for(j = 0; j < 4; ++j) // i is response position and keeps its previous value, j is loop position
        {
                if(colNum[j])
                        firstSeen = 1;
                if(firstSeen)
                {
                        rowColResponse[i] = colNum[j];
                        ++i;
                        ++colsSent;
                }
        }
        rowColResponse[i] = 'R';

        ps2_insert_to_buffer(rowColResponse, rowsSent+colsSent+4);
}

void CSIpushCursor()
{
        if(CSIcursorStackPointer < CURSOR_STACK_SIZE-1)
        {
                CSIcursorStackPointer++;
                cursorStack[0][0][CSIcursorStackPointer] = cursorRow;
                cursorStack[0][1][CSIcursorStackPointer] = cursorCol;
        }
}
void CSIpopCursor()
{
        remCursorAttr();
        if(CSIcursorStackPointer >= 0)
        {
                cursorRow = cursorStack[0][0][CSIcursorStackPointer];
                cursorCol = cursorStack[0][1][CSIcursorStackPointer];
                CSIcursorStackPointer--;
        }
        addCursorAttr();
}
void ESCpushCursor()
{
        if(ESCcursorStackPointer < CURSOR_STACK_SIZE-1)
        {
                ESCcursorStackPointer++;
                cursorStack[0][0][ESCcursorStackPointer] = cursorRow;
                cursorStack[0][1][ESCcursorStackPointer] = cursorCol;
        }
}
void ESCpopCursor()
{
        remCursorAttr();
        if(ESCcursorStackPointer >= 0)
        {
                cursorRow = cursorStack[0][0][ESCcursorStackPointer];
                cursorCol = cursorStack[0][1][ESCcursorStackPointer];
                ESCcursorStackPointer--;
        }
        addCursorAttr();
}

////////////////// Screen Manipulation //////////////////////

void bufClear(void)
{
        int i,j;
        for(i = 0; i < ROWS; ++i)
        {
                for(j = 0; j < COLS; ++j)
                {
                        screenChars[i][j] = ' ';
                        screenFgColor[i][j] = Grey;
                        screenBgColor[i][j] = Black;
                        screenDisplayOps[i][j] = 0;
                }
        }
}


void flushScreen()
{
        int i,j;
        int Io, In; // Translated i for screen top differences due to scrolling
        char letter;
        static int count = 0;
        static unsigned int refreshes = 0;
        #define REFRESH_COUNT 20
        clearExtraneousCursors();
        count = (count+1) % (REFRESH_COUNT +1);
        if (count == REFRESH_COUNT)
                refreshes++; // count number of refreshes, for blinking

        for(i = ROWS-1; i >= 0; --i) // Start from the bottom of the screen
        {
                Io = (i + oldScreenTop) % ROWS;
                In = (i + screenTop) % ROWS;
```

```c
                for(j = 0; j < COLS; ++j)
                {
                        if (oldScreenChars[Io][j] != screenChars[In][j]
                                || oldScreenFgColor[Io][j] != screenFgColor[In][j]
                                || oldScreenBgColor[Io][j] != screenBgColor[In][j]
                          || oldScreenDisplayOps[Io][j] != screenDisplayOps[In][j]
                                || count == REFRESH_COUNT) // redraw everything every so often to clear out glitches
                        {
                                letter = screenChars[In][j];
                                if(screenDisplayOps[In][j] & BLINK && refreshes%2)
                                { // blink every other whole screen refresh cycle
                                        letter = ' ';
                                }
                                if ((screenDisplayOps[In][j] & CURSOR_CHAR) && drawCursor || screenDisplayOps[In][j] & REVERSE_VIDEO)
                                {
                                        LCD_DrawChar_rc (i, j, screenBgColor[In][j], screenFgColor[In][j], letter, screenDisplayOps[In][j] & UNDERLINE);
                                }
                                else
                                        LCD_DrawChar_rc (i, j, screenFgColor[In][j], screenBgColor[In][j], letter, screenDisplayOps[In][j] & UNDERLINE);
                        }
                }
        }

        for(i = 0; i < ROWS; ++i)
        {
                for (j = 0; j < COLS; ++j)
                {
                        oldScreenChars[i][j] = screenChars[i][j];
                        oldScreenFgColor[i][j] = screenFgColor[i][j];
                        oldScreenBgColor[i][j] = screenBgColor[i][j];
                }
        }
        oldScreenTop = screenTop;


}

////////////////// Main Handler ///////////////////////////////

void handleAscii(unsigned char *buf, int bytes)
// Outputs ascii to screen or handles escape codes
// Much of the escape code handling is in "man console_codes" in in the Linux Programmer's Manual
{
        int i,j,k, rowTr;
        static unsigned int escStat = 0; // variable to hold current escape sequence status
        #define ESC_ON 0x00000001
        #define CSI_ON 0x00000002
        #define CSI_QUESTION_ON 0x00000004
        // TODO - add flags for multi-stage escapes
        #define CSI_NUM_MAX 25
        static unsigned int csi_nums[CSI_NUM_MAX];
        static unsigned int csi_nums_rec = 0;
        static unsigned int csi_digits = 0; // to hold digits until a separator is hit

        keepCursorInBounds();
        for(i = 0; i < bytes; ++i)
        {
                if(escStat & CSI_ON)
                {
                        // handle csi codes
                        // if you get:
                        // <CSI escape><number>m
                        // you change colors.  You can add multiple color options like this:
                        // <CSI esc><number>;<number>m
                        // Other CSI sequences behave similarly, a semicolon separated list of decimal numbers
                        // The function is determined by the terminating character (m for color options, others for other things)
                        // Note that the decimal number can be multiple bytes long... but probably not more than 2.
                        // My testing on the Linux console shows that numbers with more than 2 digits are to be simply ignored.

                        if (buf[i] >= 48 && buf[i] <= 57)
                        // numbers in ascii are 48-57 (0-9)
                        {
                                csi_digits *= 10; // decimal left shift
                                csi_digits += buf[i] - 48; // add in the new number
                        }
                        else if (buf[i] == ';')
                        { // separator for another number
                                if (csi_nums_rec < CSI_NUM_MAX)
                                {
                                        csi_nums[csi_nums_rec++] = csi_digits;
                                }
                                csi_digits = 0;
                        }
                        else if (buf[i] == '?')
                        {
                                escStat |= CSI_QUESTION_ON;
                        }
                        else
                        { // Here we handle the numbers received based on the terminating character.
                                // if no numbers have been received, treat it as if a 0 were received
                                csi_nums[csi_nums_rec++] = csi_digits;
                                csi_digits = 0;

                                switch (buf[i])
                                {
                                        case 'm': // Colors!!!
                                                handle_colorCodes(csi_nums, csi_nums_rec);
                                                break;
                                        case 'A': // move cursor up n rows
                                                for(j = 0; j < csi_nums_rec; ++j)
                                                {
                                                        if(csi_nums[j] == 0)
                                                                csi_nums[j] = 1;
                                                        moveCursor(csi_nums[j], UP);
                                                }
                                                break;
                                        case 'B': // move down n rows
                                        case 'e': // same as B
                                                for(j = 0; j < csi_nums_rec; ++j)
                                                {
                                                        if(csi_nums[j] == 0)
                                                                csi_nums[j] = 1;
                                                        moveCursor(csi_nums[j], DOWN);
                                                }
                                                break;
                                        case 'C': // move right n rows
                                        case 'a': // same as C
                                                for(j = 0; j < csi_nums_rec; ++j)
                                                {
                                                        if(csi_nums[j] == 0)
                                                                csi_nums[j] = 1;
                                                        moveCursor(csi_nums[j], RIGHT);
                                                }
                                                break;
                                        case 'D': // move left n rows
                                                for(j = 0; j < csi_nums_rec; ++j)
                                                {
                                                        if(csi_nums[j] == 0)
                                                                csi_nums[j] = 1;
                                                        moveCursor(csi_nums[j], LEFT);
                        }
                        break;
                    case 'E': // move cursor down n rows, to column 1
                        for(j = 0; j < csi_nums_rec; ++j)
                        {
                            if(csi_nums[j] == 0)
                                csi_nums[j] = 1;
                            moveCursor(csi_nums[j], DOWN);
                        }
                        cursorCol = 0;
                        break;
                    case 'F': // move cursor up n rows, to column 1
                        for(j = 0; j < csi_nums_rec; ++j)
                        {
                            if(csi_nums[j] == 0)
                                csi_nums[j] = 1;
                            moveCursor(csi_nums[j], UP);
                        }
                        cursorCol = 0;
                        break;
                    case 'G': // move cursor to indicated column, current row
                        remCursorAttr();
                        cursorCol = csi_nums[csi_nums_rec-1] - 1;
                        keepCursorInBounds();
                        addCursorAttr();
                        break;
                    case 'H':  // move cursor to indicated row,column
                    case 'f': // same as H
                        remCursorAttr();
                        if (csi_nums_rec <2)
                            cursorCol = 0;
                        else
                            cursorCol = csi_nums[1] -1;
                        cursorRow = csi_nums[0] -1;

                        keepCursorInBounds();
                        addCursorAttr();
                        break;
                    case 'd': // move to indicated row, current column
                        remCursorAttr();
                        cursorRow = csi_nums[csi_nums_rec-1] -1;
                        if (cursorRow < 0)
                            cursorRow = 0;
                        else if (cursorRow > ROWS - 1)
                            cursorRow = ROWS-1;
                        addCursorAttr();
                        break;
                    case 'J':
                        // Erase display
                        if (csi_nums[csi_nums_rec-1] == 1)
                        { // erase from start to cursor
                            for(j = 0; j < cursorRow; ++j)
                            {
                                rowTr = (j + screenTop) % ROWS;
                                for(k = 0; k < COLS; ++k)
                                {
                                    screenChars[rowTr][k] = ' ';
                                }
                            }
                            for(j = 0; j < cursorCol; ++j)
                            {
                                screenChars[rowTr+1][j] = ' ';
                            }
                        }
                        if (csi_nums[csi_nums_rec-1] == 0)
                        { // erase from cursor to end
                            for(j = cursorRow; j < ROWS; ++j)
                            {
                                for(k = 0; k < COLS; ++k)
                                {
                                    rowTr = (j + screenTop) % ROWS;
                                    screenChars[rowTr][k] = ' ';
                                }
                            }
                            rowTr = (cursorRow + screenTop) % ROWS;
                            for(j = 0; j < cursorCol; ++j)
                            {
                                screenChars[cursorRow][j] = ' ';
                            }
                        }
                        else if(csi_nums[csi_nums_rec-1] == 2 || csi_nums[csi_nums_rec-1] == 3)
                        { // erase whole display
                            for(j = 0; j < ROWS; ++j)
                            {
                                for (k = 0; k < COLS; ++k)
                                {
                                    screenChars[j][k] = ' ';
                                }
                            }
                        }
                        break;
                    case 'K':
                        // Erase line
                        rowTr = (cursorRow + screenTop) % ROWS;
                        if (csi_nums[csi_nums_rec-1] == 1)
                        { // erase from start to cursor
                            for(j = 0; j < cursorCol; ++j)
                            {
                                screenChars[rowTr][j] = ' ';
                            }
                        }
                        if (csi_nums[csi_nums_rec-1] == 0)
                        { // erase from cursor to end
                            for(j = cursorCol; j < COLS; ++j)
                            {
                                screenChars[rowTr][j] = ' ';
                            }
                        }
                        else if(csi_nums[csi_nums_rec-1] == 2 || csi_nums[csi_nums_rec-1] == 3)
                        { // erase whole line
                            for(j = 0; j < COLS; ++j)
                            {
                                screenChars[rowTr][j] = ' ';
                            }
                        }
                        break;
                    case 's': // push cursor position
                        CSIpushCursor();
                        break;
                    case 'u': // pop cursor position
                        CSIpopCursor();
                        break;
                    case 'n': // if we get 6, it reports cursor position
                        if (csi_nums[0] == 6)
                        {
                            reportCursorToHost();
                        }
                        break;
                    case 'l': // sometimes hides the cursor (ESC[?25l)
                        if (csi_nums[0] == 25 && (escStat & CSI_QUESTION_ON))
                        {
                            drawCursor = 0;
                        }
                        break;
```

```
                                        case 'h': // sometimes shows the cursor (ESC[?25h)
                                                if (csi_nums[0] == 25 && (escStat & CSI_QUESTION_ON))
                                                {
                                                        drawCursor = 1;
                                                }
                                                break;
                                        default:
                                                break;
                                }
                                escStat &= ~(CSI_ON | CSI_QUESTION_ON);
                                csi_nums_rec = 0;
                        }
                } //////////////// END HANDLING CSI CODES
                else if(escStat & ESC_ON)
                {
                        // handle escape codes
/*
                        The following is an exerpt from the "console_codes" man page.
                        We're not implementing all of them (Operating system command?  Obviously that's Linux only.)


                        ESC- but not CSI-sequences

       ESC c     RIS      Reset.
       ESC D     IND      Linefeed.
       ESC E     NEL      Newline.
       ESC H     HTS      Set tab stop at current column.
       ESC M     RI       Reverse linefeed.
       ESC Z     DECID    DEC private identification. The kernel returns the
                          string  ESC [ ? 6 c, claiming that it is a VT102.
       ESC 7     DECSC    Save   current    state    (cursor    coordinates,
                          attributes, character sets pointed at by G0, G1).
       ESC 8     DECRC    Restore state most recently saved by ESC 7.
       ESC [     CSI      Control sequence introducer
       ESC %              Start sequence selecting character set
       ESC % @               Select default (ISO 646 / ISO 8859-1)
       ESC % G               Select UTF-8
       ESC % 8               Select UTF-8 (obsolete)
       ESC # 8   DECALN   DEC screen alignment test - fill screen with E's.
       ESC (              Start sequence defining G0 character set
       ESC ( B               Select default (ISO 8859-1 mapping)
       ESC ( 0               Select VT100 graphics mapping
       ESC ( U               Select null mapping - straight to character ROM
       ESC ( K               Select user mapping - the map that is loaded by
                             the utility mapscrn(8).
       ESC )              Start sequence defining G1
                          (followed by one of B, 0, U, K, as above).
       ESC >     DECPNM   Set numeric keypad mode
       ESC =     DECPAM   Set application keypad mode
       ESC ]     OSC      (Should  be:  Operating  system  command)  ESC ] P
                          nrrggbb: set palette, with parameter  given  in  7
                          hexadecimal  digits after the final P :-(.  Here n
                          is the color  (0-15),  and  rrggbb  indicates  the
                          red/green/blue  values  (0-255).   ESC  ] R: reset
                          palette
*/
                        switch(buf[i])
                        {
                                case '[':
                                        escStat |= CSI_ON;
                                        break;
                                case 'c':
                                        // reset
                                        bufClear();
                                        break;
                                case '7':
                                        ESCpushCursor();
                                        break;
                                case '8':
                                        ESCpopCursor();
                                        break;
                                case 'D':
                                case 'E':
                                        handle_LF();
                                        break;
                                case 'M':
                                        // reverse line feed...
                                        moveCursor(1,UP);
                                        break;
                                case 'H':
                                        // set tab stop at current column...
                                        break;
                                case 'Z':
                                        // report DECID
                                        break;
                                default:
                                        break;
                        }

                        escStat &= ~ESC_ON;
                }
                else
                {
                        switch(buf[i])
                        {
                                // Handle control characters
                                //00 (NUL), 07 (BEL), 08 (BS), 09 (HT), 0a (LF), 0b (VT), 0c (FF),
                                //   0d (CR), 0e (SO), 0f (SI), 18 (CAN), 1a (SUB), 1b (ESC), 7f (DEL)
                                //  0x9B (CSI)
                                case 0x00: // NUL
                                        break; // it's ignored.
                                case 0x07: // BEL
                                        DAC_beep();
                                        break;
                                case 0x08: // BS
                                        // This may be tricky... we may need so implement some sort of line discipline stuff to know how many characters the user's input since the last <return>
                                        // It works properly if user input doesn't make it go on to a second line... the same problem that plagues many
                                        // X terminal emulators...
                                        if (cursorCol > 0)
                                                --cursorCol;
                                        keepCursorInBounds();
                                        break;
                                case 0x09: // HT - goes to the next tab stop or to the end of the line if there is no earlier tab stop
                                        break;
                                case 0x0A: // LF - down a line.
                                case 0x0B: // VT, same as LF
                                case 0x0C: // FF, same as VT and LF
                                        handle_LF();
                                        break;
                                case 0x0D: // CR - push the carriage to the left! On old typewriters.
                                        handle_CR();
                                        break;
                                case 0x0E: // SO - activate G1 character set
                                        break;
                                case 0x0F: // SI - activate G0 character set
                                        break;
                                case 0x18: // CAN - interrupt escape sequence (not sure what it does)
                                        break;
                                case 0x1A: // SUB - interrupt escape sequence (not sure what it does)
                                        break;
                                case 0x1B: // ESC - start escape sequence
```

```
                                        escStat |= ESC_ON;
                                        break;
                                case 0x7F: // DEL - ignored
                                        break;
                                case 0x9B: // CSI - start CSI sequence
                                        escStat |= CSI_ON;
                                        break;
                                // Normal character handling!
                                default:
                                        handle_normal(buf[i]);
                                        break;
                        }
                }
        }
}

// usart2.c

#include "usart2.h"


/*
        ; baud rate, 0x08, 12bit mantissa plus 4 bits fractional
; BRD = 8e6/(16*9600)= 52.083
  ; integer portion: int(52.083)=52 = 0b00110100      (0xD0 if 32Mhz, 0x138 for 48MHz)
  ; fractional portion: int(0.083*2^6+0.5)=6 = 0b0110  (0x32 if 32Mhz, 0x5 for 48MHz)
  ; to put in register: 0b001101000110 = 0x0341         (0xD03 if 32Mhz, 0x1385 for 48MHz)
BAUD_RATE  DCD 0x0341

; control register 1, 0x0C, bits 31-14 reserved (0), 0b10000000001100 = 0x200C   no interrupts, no break, active receiver, RX and TX enabled
USART_CTRL1_SETTINGS DCD 0x200C

        SETUPA  DCD 0x000A8AA8  ; Usart alt function settings -- see usart comments below
        ; alternate functionality pins
;USART2_CTS PA0 PD3  - input pull-up (8)
;USART2_RTS PA1 PD4  - alt push-pull  (A)
;USART2_TX PA2 PD5  - alt push-pull
;USART2_RX PA3 PD6  - input pull up
;USART2_CK PA4 PD7  - alt push-pull
        */

unsigned char usart2_data[USART2_DATA_SIZE];
int usart2_bytes_rec = 0;

void usart2_init(void)
{
        RCC->APB1ENR |= RCC_APB1ENR_USART2EN;
        //GPIOA->CRL = (GPIOA->CRL & 0xFFF00000) | 0x000A8AA8;
        GPIOA->CRL = (GPIOA->CRL & 0xFFF00000) | 0x000B8BB8;

        //USART2->BRR = 0x0341;
        USART2->BRR = 0x1385;
        USART2->CR1 = 0x202C; //Enable RXNEIE(bit 5) bit for receive interrupt
        NVIC->ISER[1] = NVIC_ISER_SETENA_6;

}

void usart2_tx(unsigned char byte)
{
        while(!(USART2->SR & USART_SR_TC));

        USART2->DR = byte;
}


void USART2_IRQHandler()
{
        if (usart2_bytes_rec < USART2_DATA_SIZE)
        {
```

```c
                usart2_data[usart2_bytes_rec] = USART2->DR;
                usart2_bytes_rec++;
        }

        NVIC->ISER[1] = NVIC_ICPR_CLRPEND_6;
}

int usart2_memcpy(unsigned char * dst)
// Copies the usart2 buffer, then resets the usart2 buffer
{
        int ret, i;

        // turn off interrupts
        NVIC->ICER[1] = NVIC_ICER_CLRENA_6;

        // copy buffer
        for(i = 0; i < usart2_bytes_rec; ++i)
        {
                dst[i] = usart2_data[i];
        }
        ret = usart2_bytes_rec;
        usart2_bytes_rec = 0;

        // enable interrupts
        NVIC->ISER[1] = NVIC_ISER_SETENA_6;

        return ret;
}
```