

Serial Terminal

Design Documentation

William Hatch and Scott Sorensen

April 2013

Table of Contents

1. Introduction	2
2. Scope	2
3. Design Overview	2
3.1 Requirements	2
3.2 Theory of Operation	3
4. Design Detail	3
4.1 Hardware Design	3
4.2 Software Design	4
4.2.1 PS/2 Keyboard Communication	4
4.2.2 USART Communication	5
4.2.3 LCD	5
4.2.4 Control Codes and Escape Sequences	6
4.2.5 Visual Effects	6
4.2.6 Onboard DAC	6
5. Verification	6
6. Conclusion	7
Appendix A: Hardware Schematic	8
Appendix B: Software Flow Charts	9
Appendix C: Verification Pictures	11
Appendix D: Source Code	12

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 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

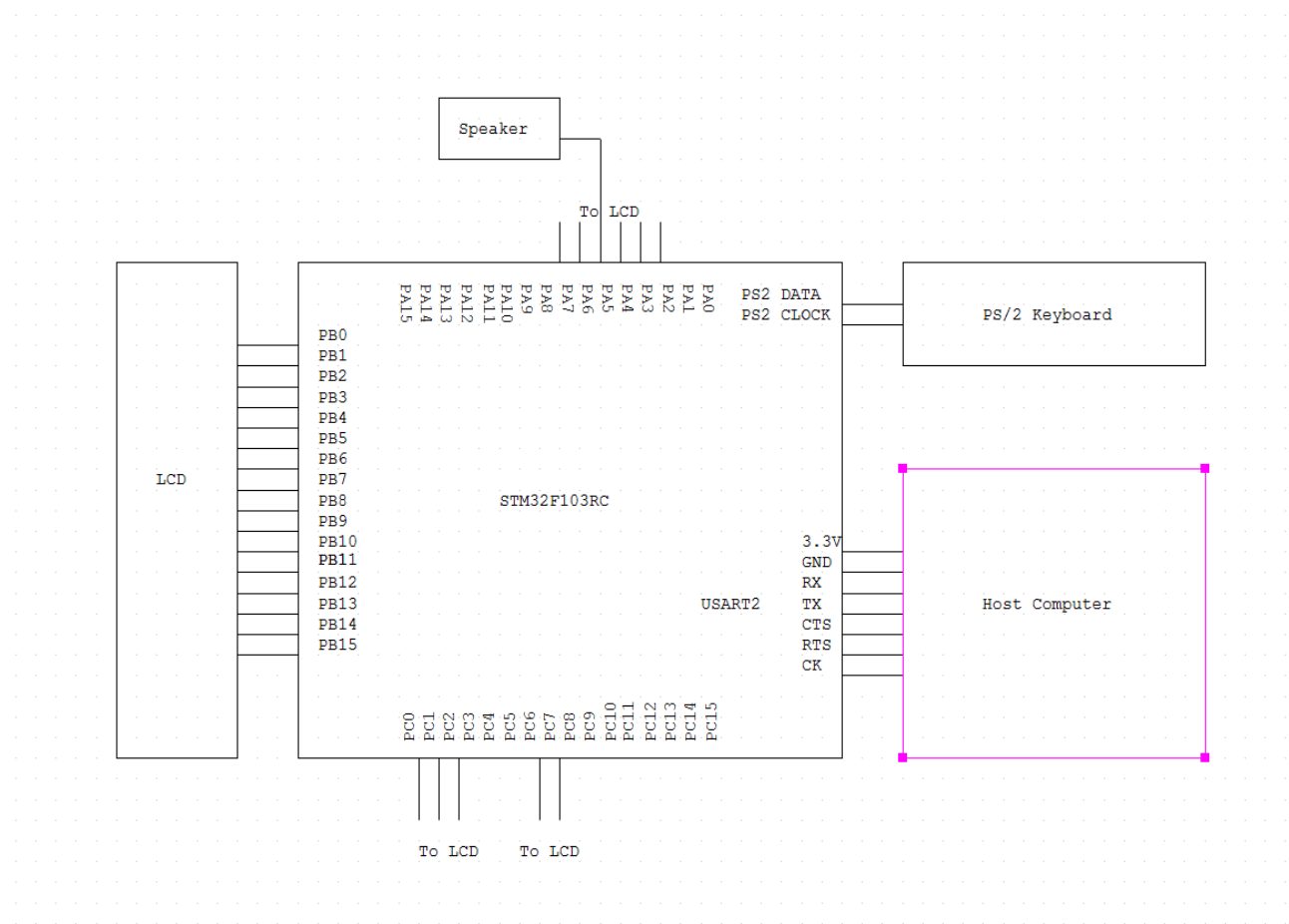
The system was hand tested to see that it received, transferred, and displayed data properly. A color comparison script was used to display all possible color combinations, and an effects script was written to test blinking, underlining, and reverse-video effects. Cursor manipulation was tested by viewing 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.

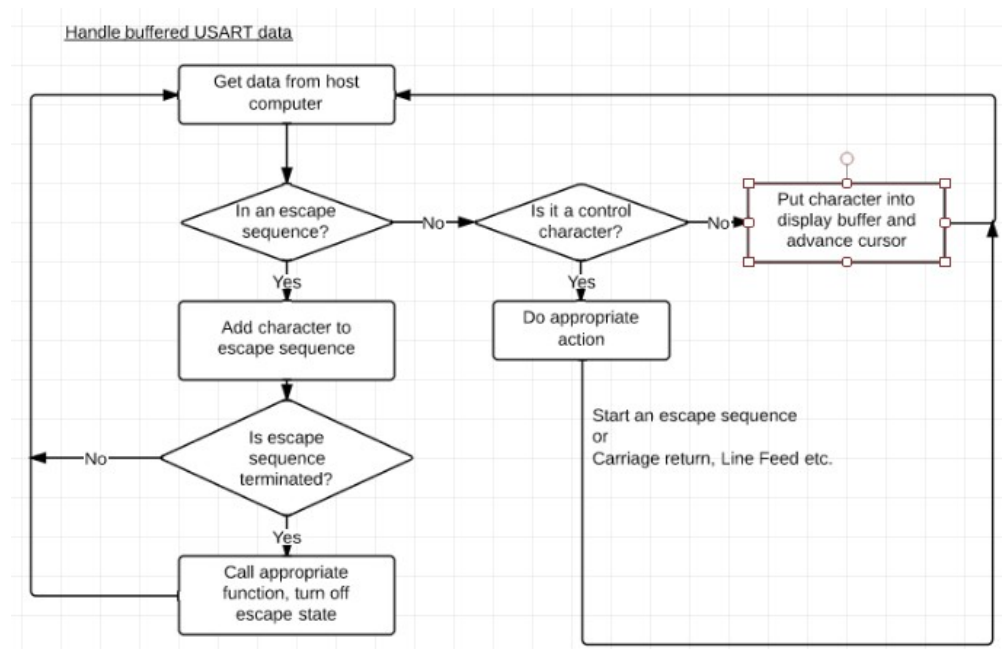
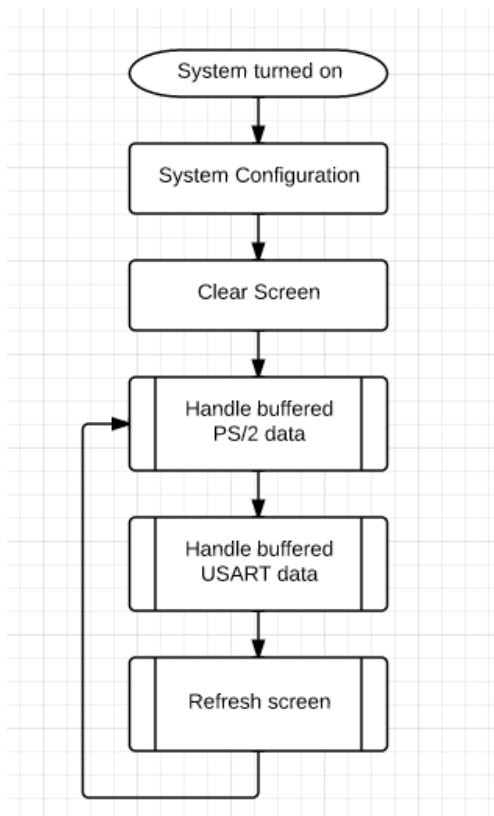
6. Conclusion

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.

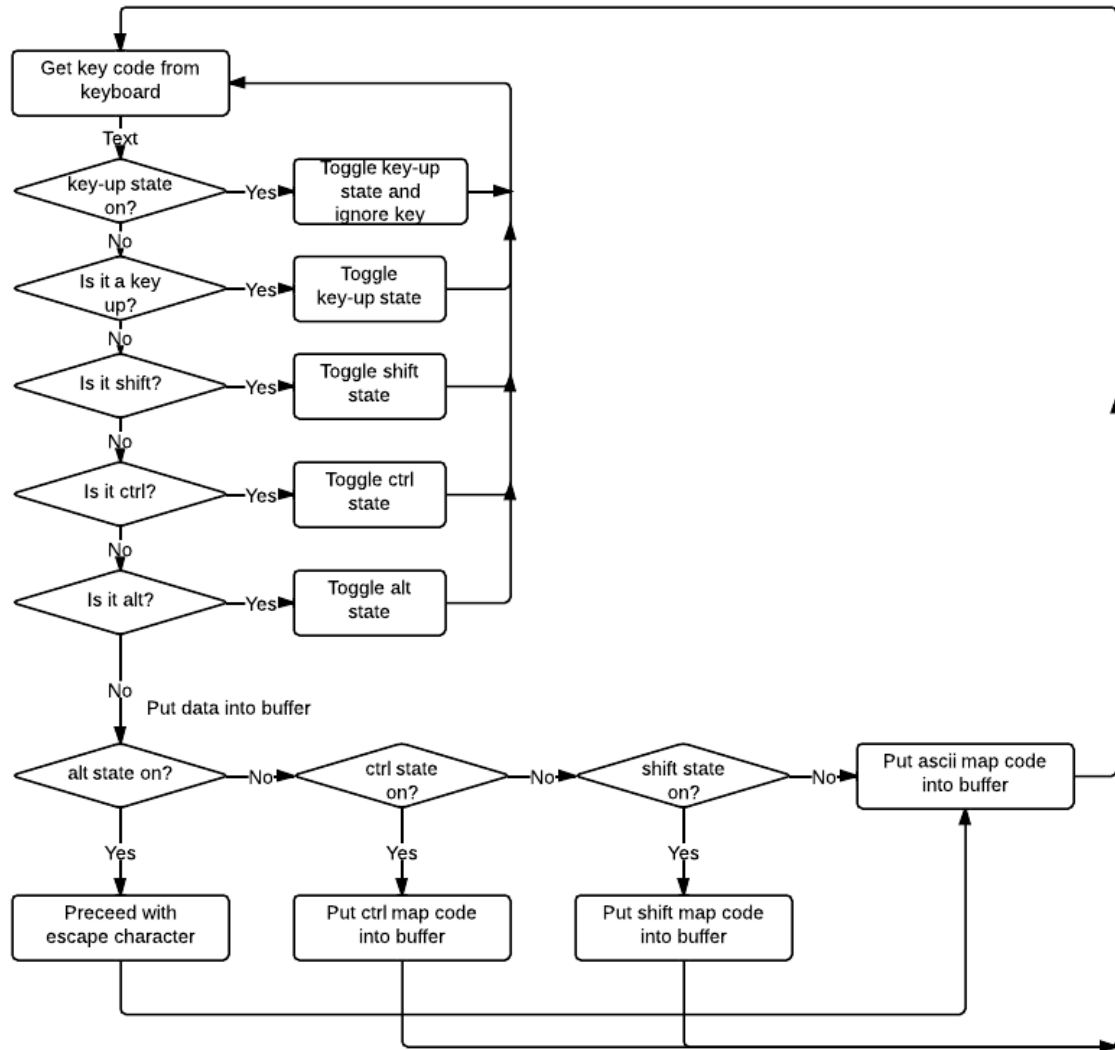
Appendix A: Hardware Schematic



Appendix B: Software Flow Charts

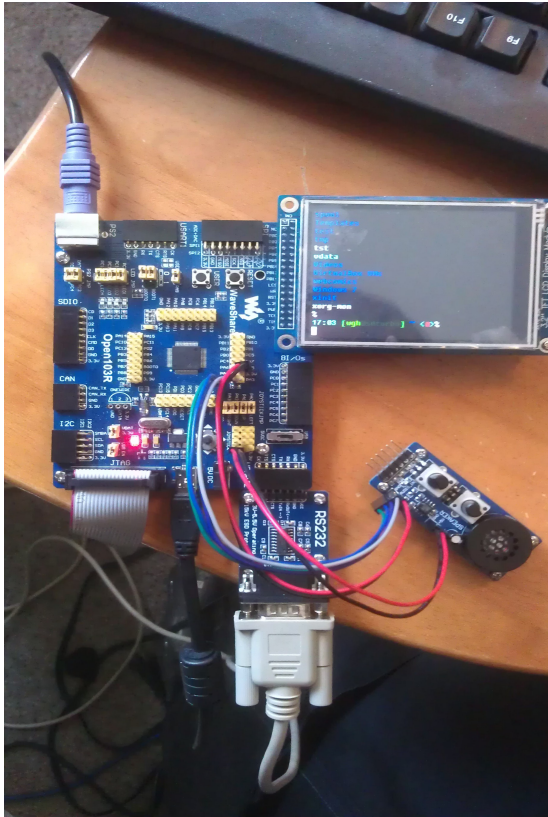


Handle buffered PS/2 data

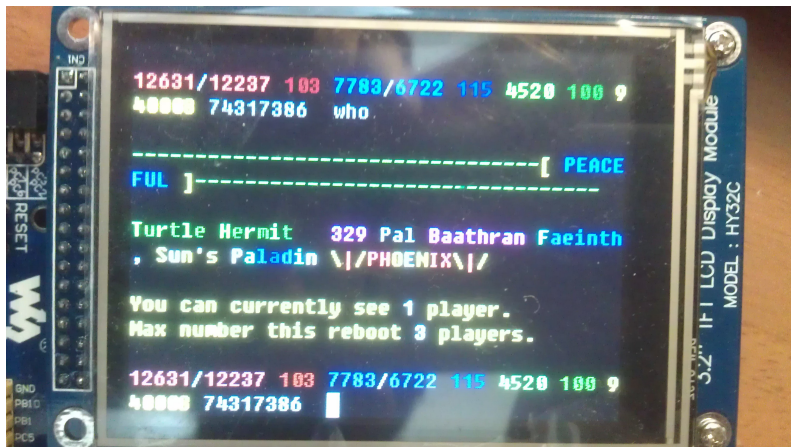


Appendix C: Verification Photographs

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



Connected to online multiplayer text-based game.



Appendix D: Source Code

```

1  /*
2  *   ECE 3710 Lab 6: ascii.h
3  *
4  *
5  */
6
7  #ifndef __ASCII_H
8  #define __ASCII_H
9
10 #include <string.h>
11
12 void get_ascii( unsigned char *, unsigned char );
13
14 #endif
15
16 /* END OF FILE */
17 //code_to_ascii.h
18
19 #ifndef CODE_TO_ASCII
20 #define CODE_TO_ASCII
21
22 void scan_code_init(void);
23
24 #endif
25 // dac.h
26
27 void DAC_init(void);
28 void DAC_beep(void);
29 //
30 //  lcd.h
31 //  ECE 3710 Microcontroller H&S
32 //  Utah State University
33 //
34
35 #ifndef __LCD_H
36 #define __LCD_H
37
38 #include "stm32f10x.h"
39
40 #define DISP_ORIENTATION 90
41
42 #if ( DISP_ORIENTATION == 90 ) || ( DISP_ORIENTATION == 270 )
43
44 #define MAX_X 320
45 #define MAX_Y 240
46 // #define CHARS_HORIZ_ON_Y 30
47 #define CHARS_HORIZ_ON_Y 28
48 #define CHARS_VERT_ON_Y 20
49 #define CHARS_HORIZ_ON_X 40
50 #define CHARS_VERT_ON_X 15
51
52 #elif ( DISP_ORIENTATION == 0 ) || ( DISP_ORIENTATION == 180 )
53
54 #define MAX_X 240
55 #define MAX_Y 320
56 #define CHARS_HORIZ_ON_Y 40
57 #define CHARS_VERT_ON_Y 15
58
59 #endif
60
61
62 /* some LCD colors */
63 #define White      0xFFFF
64 #define Black      0x0000
65 #define Grey       0xF7DE
66 #define Blue       0x001F
67 #define Blue2      0x051F
68 #define Red        0xF800
69 #define Magenta    0xF81F
70 #define Green      0x07E0

```

```

71 #define Cyan          0x7FFF
72 #define Yellow        0xFFE0
73
74
75 #define TermBlack Black
76 #define TermBlackBright 0x52aa
77 #define TermRed 0xa800
78 #define TermRedBright 0xfaaa
79 #define TermGreen 0x540
80 #define TermGreenBright 0x57ea
81 #define TermBrown 0xaaa0
82 #define TermBrownBright 0xffea
83 #define TermBlue 0x15
84 #define TermBlueBright 0x52bf
85 #define TermMagenta 0xa815
86 #define TermMagentaBright 0xfabf
87 #define TermCyan 0x555
88 #define TermCyanBright 0x57ff
89 #define TermWhite 0xa554
90 #define TermWhiteBright White
91 #define TermDefault TermWhite
92 #define TermDefaultBright TermWhiteBright
93
94
95 void LCD_Config(void);
96 void LCD_Initialization(void);
97 void LCD_Clear( unsigned short Color );
98
99 void LCD_WriteIndex( unsigned short index );
100 void LCD_WriteData( unsigned short data );
101 void LCD_Write_Generic(unsigned short toWrite, unsigned short dataBool);
102 void LCD_WriteReg( unsigned short LCD_Reg, unsigned short LCD_RegValue );
103
104 void LCD_SetCursor( unsigned short x, unsigned int y );
105 void delay_ms( unsigned int ms );
106
107 void LCD_DrawSquare( unsigned short x, unsigned short y, unsigned short h, unsigned short w,
    unsigned short color );
108
109
110 void LCD_DrawCharacterOnY (unsigned short x, unsigned short y, unsigned short fgColor, unsigned
    short bgColor, unsigned char symbol);
111 void LCD_WriteCharactersOnY (unsigned short x, unsigned short y, unsigned short fgColor, unsigned
    short bgColor, char* words, int maxLength);
112 void LCD_WriteLinesOnY(unsigned short x, unsigned short fgColor, unsigned short bgColor, char*
    words, char drawToLineEnd);
113
114 void LCD_DrawChar_rc (unsigned int row, unsigned int col, unsigned short fgColor, unsigned short
    bgColor, unsigned char symbol, unsigned char underline);
115
116 #endif
117
118 // END OF FILE
119 //ps2_over_gpioc.h
120 #ifndef PS2_OVER_GPIOC
121 #define PS2_OVER_GPIOC
122
123 #define PS2_DATA_SIZE 1000
124
125
126
127 void ps2_over_gpioc_init(void);
128 //void ps2_dump_data_over_usart2(void);
129 int ps2_memcpy(unsigned char * dst);
130 void ps2_insert_to_buffer(char *insert, int size);
131
132
133 #endif
134
135

```

```

136 // terminal.h
137 #ifndef __TERMINAL
138 #define __TERMINAL
139
140 #include "lcd.h"
141
142 #define ROWS CHARS_VERT_ON_X // Num lines
143 #define COLS CHARS_HORIZ_ON_X // Num columns
144
145
146
147 void flushScreen(void);
148 void bufClear(void);
149 void handleAscii(unsigned char *buf, int bytes);
150
151 #endif
152
153 // usart2.h
154
155 #ifndef __USART2
156 #define __USART2
157 #include "stm32f10x.h"
158
159
160 #define USART2_DATA_SIZE 1000
161
162 void usart2_init(void);
163
164 void usart2_tx(unsigned char byte);
165 int usart2_memcpy(unsigned char * dst);
166
167 #endif
168 /*
169  * ECE 3710 Lab 6: ascii.c
170  *
171  */
172
173 #include "ascii.h"
174
175 static const unsigned char ascii[95][16] =
176 {{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}, /* " " */
177 {0x00,0x00,0x00,0x18,0x3C,0x3C,0x3C,0x18,0x18,0x00,0x18,0x18,0x00,0x00,0x00,0x00}, /* "!" */
178 {0x00,0x00,0x00,0x66,0x66,0x66,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}, /* "\" */
179 {0x00,0x00,0x00,0x36,0x36,0x7F,0x36,0x36,0x36,0x7F,0x36,0x36,0x00,0x00,0x00,0x00}, /* "#" */
180 {0x00,0x18,0x18,0x3C,0x66,0x60,0x30,0x18,0x0C,0x06,0x66,0x3C,0x18,0x18,0x00,0x00}, /* "$" */
181 {0x00,0x00,0x70,0xD8,0xDA,0x76,0x0C,0x18,0x30,0x6E,0x5B,0x1B,0x0E,0x00,0x00,0x00}, /* "%" */
182 {0x00,0x00,0x00,0x38,0x6C,0x6C,0x38,0x60,0x6F,0x66,0x66,0x3B,0x00,0x00,0x00,0x00}, /* "&" */
183 {0x00,0x00,0x00,0x18,0x18,0x18,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}, /* "' " */
184 {0x00,0x00,0x00,0x0C,0x18,0x18,0x30,0x30,0x30,0x30,0x18,0x18,0x0C,0x00,0x00,0x00}, /* "(" */
185 {0x00,0x00,0x00,0x30,0x18,0x18,0x0C,0x0C,0x0C,0x0C,0x0C,0x18,0x18,0x30,0x00,0x00}, /* ")" */
186 {0x00,0x00,0x00,0x00,0x00,0x36,0x1C,0x7F,0x1C,0x36,0x00,0x00,0x00,0x00,0x00,0x00}, /* "*" */
187 {0x00,0x00,0x00,0x00,0x00,0x18,0x18,0x7E,0x18,0x18,0x00,0x00,0x00,0x00,0x00,0x00}, /* "+" */
188 {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x1C,0x1C,0x0C,0x18,0x00,0x00}, /* ", " */
189 {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x7E,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}, /* "- " */
190 {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x1C,0x1C,0x00,0x00,0x00,0x00}, /* ". " */
191 {0x00,0x00,0x00,0x06,0x06,0x0C,0x0C,0x18,0x18,0x30,0x30,0x60,0x60,0x00,0x00,0x00}, /* "/" */
192 {0x00,0x00,0x00,0x1E,0x33,0x37,0x37,0x33,0x3B,0x3B,0x33,0x1E,0x00,0x00,0x00,0x00}, /* "0" */
193 {0x00,0x00,0x00,0x0C,0x1C,0x7C,0x0C,0x0C,0x0C,0x0C,0x0C,0x0C,0x00,0x00,0x00,0x00}, /* "1" */
194 {0x00,0x00,0x00,0x3C,0x66,0x66,0x06,0x0C,0x18,0x30,0x60,0x7E,0x00,0x00,0x00,0x00}, /* "2" */
195 {0x00,0x00,0x00,0x3C,0x66,0x66,0x06,0x1C,0x06,0x66,0x66,0x3C,0x00,0x00,0x00,0x00}, /* "3" */
196 {0x00,0x00,0x00,0x30,0x30,0x36,0x36,0x36,0x66,0x7F,0x06,0x06,0x00,0x00,0x00,0x00}, /* "4" */
197 {0x00,0x00,0x00,0x7E,0x60,0x60,0x60,0x7C,0x06,0x06,0x0C,0x78,0x00,0x00,0x00,0x00}, /* "5" */
198 {0x00,0x00,0x00,0x1C,0x18,0x30,0x7C,0x66,0x66,0x66,0x66,0x3C,0x00,0x00,0x00,0x00}, /* "6" */
199 {0x00,0x00,0x00,0x7E,0x06,0x0C,0x0C,0x18,0x18,0x30,0x30,0x30,0x00,0x00,0x00,0x00}, /* "7" */
200 {0x00,0x00,0x00,0x0C,0x66,0x66,0x76,0x3C,0x6E,0x66,0x66,0x3C,0x00,0x00,0x00,0x00}, /* "8" */
201 {0x00,0x00,0x00,0x3C,0x66,0x66,0x66,0x66,0x3E,0x0C,0x18,0x38,0x00,0x00,0x00,0x00}, /* "9" */
202 {0x00,0x00,0x00,0x00,0x00,0x1C,0x1C,0x00,0x00,0x00,0x1C,0x1C,0x00,0x00,0x00,0x00}, /* ":" */
203 {0x00,0x00,0x00,0x00,0x00,0x1C,0x1C,0x00,0x00,0x00,0x1C,0x1C,0x0C,0x18,0x00,0x00}, /* "; " */
204 {0x00,0x00,0x00,0x06,0x0C,0x18,0x30,0x60,0x30,0x18,0x0C,0x06,0x00,0x00,0x00,0x00}, /* "<" */
205 {0x00,0x00,0x00,0x00,0x00,0x00,0x7E,0x00,0x7E,0x00,0x00,0x00,0x00,0x00,0x00,0x00}, /* "=" */

```



```

206 {0x00,0x00,0x00,0x60,0x30,0x18,0x0C,0x06,0x0C,0x18,0x30,0x60,0x00,0x00,0x00,0x00}, /* ">" */
207 {0x00,0x00,0x00,0x3C,0x66,0x66,0x0C,0x18,0x18,0x00,0x18,0x18,0x00,0x00,0x00,0x00}, /* "?" */
208 {0x00,0x00,0x00,0x7E,0xC3,0xC3,0xCF,0xDB,0xDB,0xCF,0xC0,0x7F,0x00,0x00,0x00,0x00}, /* "@" */
209 {0x00,0x00,0x00,0x18,0x3C,0x66,0x66,0x66,0x7E,0x66,0x66,0x66,0x00,0x00,0x00,0x00}, /* "A" */
210 {0x00,0x00,0x00,0x7C,0x66,0x66,0x66,0x7C,0x66,0x66,0x66,0x7C,0x00,0x00,0x00,0x00}, /* "B" */
211 {0x00,0x00,0x00,0x3C,0x66,0x66,0x60,0x60,0x60,0x66,0x66,0x3C,0x00,0x00,0x00,0x00}, /* "C" */
212 {0x00,0x00,0x00,0x78,0x6C,0x66,0x66,0x66,0x66,0x66,0x6C,0x78,0x00,0x00,0x00,0x00}, /* "D" */
213 {0x00,0x00,0x00,0x7E,0x60,0x60,0x60,0x7C,0x60,0x60,0x60,0x7E,0x00,0x00,0x00,0x00}, /* "E" */
214 {0x00,0x00,0x00,0x7E,0x60,0x60,0x60,0x7C,0x60,0x60,0x60,0x60,0x00,0x00,0x00,0x00}, /* "F" */
215 {0x00,0x00,0x00,0x3C,0x66,0x66,0x60,0x60,0x6E,0x66,0x66,0x3E,0x00,0x00,0x00,0x00}, /* "G" */
216 {0x00,0x00,0x00,0x66,0x66,0x66,0x66,0x7E,0x66,0x66,0x66,0x66,0x00,0x00,0x00,0x00}, /* "H" */
217 {0x00,0x00,0x00,0x3C,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x3C,0x00,0x00,0x00,0x00}, /* "I" */
218 {0x00,0x00,0x00,0x06,0x06,0x06,0x06,0x06,0x06,0x66,0x66,0x3C,0x00,0x00,0x00,0x00}, /* "J" */
219 {0x00,0x00,0x00,0x66,0x66,0x6C,0x6C,0x78,0x6C,0x6C,0x66,0x66,0x00,0x00,0x00,0x00}, /* "K" */
220 {0x00,0x00,0x00,0x60,0x60,0x60,0x60,0x60,0x60,0x60,0x7E,0x00,0x00,0x00,0x00}, /* "L" */
221 {0x00,0x00,0x00,0x63,0x63,0x77,0x6B,0x6B,0x6B,0x63,0x63,0x63,0x00,0x00,0x00,0x00}, /* "M" */
222 {0x00,0x00,0x00,0x63,0x63,0x73,0x7B,0x6F,0x67,0x63,0x63,0x63,0x00,0x00,0x00,0x00}, /* "N" */
223 {0x00,0x00,0x00,0x3C,0x66,0x66,0x66,0x66,0x66,0x66,0x3C,0x00,0x00,0x00,0x00}, /* "O" */
224 {0x00,0x00,0x00,0x7C,0x66,0x66,0x66,0x7C,0x60,0x60,0x60,0x60,0x00,0x00,0x00,0x00}, /* "P" */
225 {0x00,0x00,0x00,0x3C,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x3C,0x0C,0x06,0x00,0x00}, /* "Q" */
226 {0x00,0x00,0x00,0x7C,0x66,0x66,0x66,0x7C,0x6C,0x66,0x66,0x66,0x00,0x00,0x00,0x00}, /* "R" */
227 {0x00,0x00,0x00,0x3C,0x66,0x60,0x30,0x18,0x0C,0x06,0x66,0x3C,0x00,0x00,0x00,0x00}, /* "S" */
228 {0x00,0x00,0x00,0x7E,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x00,0x00,0x00,0x00}, /* "T" */
229 {0x00,0x00,0x00,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x3C,0x00,0x00,0x00,0x00}, /* "U" */
230 {0x00,0x00,0x00,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x3C,0x18,0x00,0x00,0x00,0x00}, /* "V" */
231 {0x00,0x00,0x00,0x63,0x63,0x63,0x6B,0x6B,0x6B,0x36,0x36,0x36,0x00,0x00,0x00,0x00}, /* "W" */
232 {0x00,0x00,0x00,0x66,0x66,0x34,0x18,0x18,0x2C,0x66,0x66,0x66,0x00,0x00,0x00,0x00}, /* "X" */
233 {0x00,0x00,0x00,0x66,0x66,0x66,0x66,0x3C,0x18,0x18,0x18,0x18,0x00,0x00,0x00,0x00}, /* "Y" */
234 {0x00,0x00,0x00,0x7E,0x06,0x06,0x0C,0x18,0x30,0x60,0x60,0x7E,0x00,0x00,0x00,0x00}, /* "Z" */
235 {0x00,0x00,0x00,0x3C,0x30,0x30,0x30,0x30,0x30,0x30,0x30,0x30,0x30,0x30,0x3C,0x00}, /* "[" */
236 {0x00,0x00,0x00,0x60,0x60,0x30,0x30,0x18,0x18,0x0C,0x0C,0x06,0x06,0x00,0x00,0x00}, /* "\" */
237 {0x00,0x00,0x00,0x3C,0x0C,0x0C,0x0C,0x0C,0x0C,0x0C,0x0C,0x0C,0x0C,0x0C,0x3C,0x00}, /* "]" */
238 {0x00,0x18,0x3C,0x66,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}, /* "^" */
239 {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xFF}, /* "_" */
240 {0x00,0x00,0x00,0x18,0x18,0x18,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}, /* "`" */
241 {0x00,0x00,0x00,0x00,0x00,0x3C,0x06,0x06,0x3E,0x66,0x66,0x3E,0x00,0x00,0x00,0x00}, /* "a" */
242 {0x00,0x00,0x00,0x60,0x60,0x7C,0x66,0x66,0x66,0x66,0x66,0x7C,0x00,0x00,0x00,0x00}, /* "b" */
243 {0x00,0x00,0x00,0x00,0x00,0x3C,0x66,0x60,0x60,0x60,0x66,0x3C,0x00,0x00,0x00,0x00}, /* "c" */
244 {0x00,0x00,0x00,0x06,0x06,0x3E,0x66,0x66,0x66,0x66,0x66,0x3E,0x00,0x00,0x00,0x00}, /* "d" */
245 {0x00,0x00,0x00,0x00,0x00,0x3C,0x66,0x66,0x7E,0x60,0x60,0x3C,0x00,0x00,0x00,0x00}, /* "e" */
246 {0x00,0x00,0x00,0x1E,0x30,0x30,0x30,0x7E,0x30,0x30,0x30,0x30,0x00,0x00,0x00,0x00}, /* "f" */
247 {0x00,0x00,0x00,0x00,0x00,0x3E,0x66,0x66,0x66,0x66,0x66,0x3E,0x06,0x06,0x7C,0x00}, /* "g" */
248 {0x00,0x00,0x00,0x60,0x60,0x7C,0x66,0x66,0x66,0x66,0x66,0x66,0x00,0x00,0x00,0x00}, /* "h" */
249 {0x00,0x00,0x18,0x18,0x00,0x78,0x18,0x18,0x18,0x18,0x18,0x7E,0x00,0x00,0x00,0x00}, /* "i" */
250 {0x00,0x00,0x0C,0x0C,0x00,0x3C,0x0C,0x0C,0x0C,0x0C,0x0C,0x0C,0x0C,0x0C,0x78,0x00}, /* "j" */
251 {0x00,0x00,0x00,0x60,0x60,0x66,0x66,0x6C,0x78,0x6C,0x66,0x66,0x00,0x00,0x00,0x00}, /* "k" */
252 {0x00,0x00,0x00,0x78,0x18,0x18,0x18,0x18,0x18,0x18,0x7E,0x00,0x00,0x00,0x00}, /* "l" */
253 {0x00,0x00,0x00,0x00,0x00,0x7E,0x6B,0x6B,0x6B,0x6B,0x6B,0x63,0x00,0x00,0x00,0x00}, /* "m" */
254 {0x00,0x00,0x00,0x00,0x00,0x7C,0x66,0x66,0x66,0x66,0x66,0x66,0x00,0x00,0x00,0x00}, /* "n" */
255 {0x00,0x00,0x00,0x00,0x00,0x3C,0x66,0x66,0x66,0x66,0x66,0x3C,0x00,0x00,0x00,0x00}, /* "o" */
256 {0x00,0x00,0x00,0x00,0x00,0x7C,0x66,0x66,0x66,0x66,0x66,0x7C,0x60,0x60,0x60,0x00}, /* "p" */
257 {0x00,0x00,0x00,0x00,0x00,0x3E,0x66,0x66,0x66,0x66,0x66,0x3E,0x06,0x06,0x06,0x00}, /* "q" */
258 {0x00,0x00,0x00,0x00,0x00,0x66,0x6E,0x70,0x60,0x60,0x60,0x60,0x00,0x00,0x00,0x00}, /* "r" */
259 {0x00,0x00,0x00,0x00,0x00,0x3E,0x60,0x60,0x3C,0x06,0x06,0x7C,0x00,0x00,0x00,0x00}, /* "s" */
260 {0x00,0x00,0x00,0x30,0x30,0x7E,0x30,0x30,0x30,0x30,0x30,0x1E,0x00,0x00,0x00,0x00}, /* "t" */
261 {0x00,0x00,0x00,0x00,0x00,0x66,0x66,0x66,0x66,0x66,0x66,0x3E,0x00,0x00,0x00,0x00}, /* "u" */
262 {0x00,0x00,0x00,0x00,0x00,0x66,0x66,0x66,0x66,0x66,0x66,0x3C,0x18,0x00,0x00,0x00}, /* "v" */
263 {0x00,0x00,0x00,0x00,0x00,0x63,0x6B,0x6B,0x6B,0x6B,0x36,0x36,0x00,0x00,0x00,0x00}, /* "w" */
264 {0x00,0x00,0x00,0x00,0x00,0x66,0x66,0x3C,0x18,0x3C,0x66,0x66,0x00,0x00,0x00,0x00}, /* "x" */
265 {0x00,0x00,0x00,0x00,0x00,0x66,0x66,0x66,0x66,0x66,0x66,0x3C,0x0C,0x18,0xF0,0x00}, /* "y" */
266 {0x00,0x00,0x00,0x00,0x00,0x7E,0x06,0x0C,0x18,0x30,0x60,0x7E,0x00,0x00,0x00,0x00}, /* "z" */
267 {0x00,0x00,0x00,0x0C,0x18,0x18,0x18,0x30,0x60,0x30,0x18,0x18,0x18,0x0C,0x00,0x00}, /* "{" */
268 {0x00,0x00,0x00,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x00}, /* "|" */
269 {0x00,0x00,0x00,0x30,0x18,0x18,0x18,0x0C,0x06,0x0C,0x18,0x18,0x18,0x30,0x00,0x00}, /* "}" */
270 {0x00,0x00,0x00,0x71,0xDB,0x8E,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}}; /* "~" */
271
272 void get_ascii( unsigned char* buffer, unsigned char chr )
273 {
274     memcpy( buffer, ascii[chr-32], 16 );
275 }

```

```

276
277 /* END OF FILE */
278 //code_to_ascii.c
279
280 #include "code_to_ascii.h"
281
282 unsigned char map[0x100];
283 unsigned char shift_map[0x100];
284 unsigned char ctl_map[0x100];
285
286 void scan_code_init(void)
287 {
288     int i = 0;
289     for (i=0; i<0x100; i++)
290     {
291         map[i] = 0;
292         shift_map[i] = 0;
293         ctl_map[i] = 0;
294     }
295
296     map[0x1C]='a';
297     map[0x32]='b';
298     map[0x21]='c';
299     map[0x23]='d';
300     map[0x24]='e';
301     map[0x2B]='f';
302     map[0x34]='g';
303     map[0x33]='h';
304     map[0x43]='i';
305     map[0x3B]='j';
306     map[0x42]='k';
307     map[0x4B]='l';
308     map[0x3A]='m';
309     map[0x31]='n';
310     map[0x44]='o';
311     map[0x4D]='p';
312     map[0x15]='q';
313     map[0x2D]='r';
314     map[0x1B]='s';
315     map[0x2C]='t';
316     map[0x3C]='u';
317     map[0x2A]='v';
318     map[0x1D]='w';
319     map[0x22]='x';
320     map[0x35]='y';
321     map[0x1A]='z';
322     map[0x45]='0';
323     map[0x16]='1';
324     map[0x1E]='2';
325     map[0x26]='3';
326     map[0x25]='4';
327     map[0x2E]='5';
328     map[0x36]='6';
329     map[0x3D]='7';
330     map[0x3E]='8';
331     map[0x46]='9';
332
333     map[0x54]='[';
334     map[0x0E]='`';
335     map[0x4E]='-';
336     map[0x55]='=';
337     map[0x5D]='\\';
338     map[0x52]='\'';
339     map[0x49]='.';
340     map[0x41]=',';
341     map[0x4C]=';';
342     map[0x5B]=']';
343     map[0x4A]='/';
344
345     map[0x29]=0x20; //SPACE

```

```

346 map[0x5A]=0x0D;//ENTER
347 map[0x76]=0x1B;//ESC
348 map[0x66]=0x08;//BKSP
349 map[0x0D]=0x09;//TAB
350
351 map[0x7C]='*';// 'KP *';
352 map[0x79]='+';// 'KP +';
353 map[0x7B]='-';// 'KP -';
354 map[0x71]='.';// 'KP .';
355 map[0x70]='0';// 'KP 0';
356 map[0x69]='1';// 'KP 1';
357 map[0x72]='2';// 'KP 2';
358 map[0x7A]='3';// 'KP 3';
359 map[0x6B]='4';// 'KP 4';
360 map[0x73]='5';// 'KP 5';
361 map[0x74]='6';// 'KP 6';
362 map[0x6C]='7';// 'KP 7';
363 map[0x75]='8';// 'KP 8';
364 map[0x7D]='9';// 'KP 9';
365
366
367 shift_map[0x1C]='A';
368 shift_map[0x32]='B';
369 shift_map[0x21]='C';
370 shift_map[0x23]='D';
371 shift_map[0x24]='E';
372 shift_map[0x2B]='F';
373 shift_map[0x34]='G';
374 shift_map[0x33]='H';
375 shift_map[0x43]='I';
376 shift_map[0x3B]='J';
377 shift_map[0x42]='K';
378 shift_map[0x4B]='L';
379 shift_map[0x3A]='M';
380 shift_map[0x31]='N';
381 shift_map[0x44]='O';
382 shift_map[0x4D]='P';
383 shift_map[0x15]='Q';
384 shift_map[0x2D]='R';
385 shift_map[0x1B]='S';
386 shift_map[0x2C]='T';
387 shift_map[0x3C]='U';
388 shift_map[0x2A]='V';
389 shift_map[0x1D]='W';
390 shift_map[0x22]='X';
391 shift_map[0x35]='Y';
392 shift_map[0x1A]='Z';
393 shift_map[0x45]=')';
394 shift_map[0x16]='!';
395 shift_map[0x1E]='@';
396 shift_map[0x26]='#';
397 shift_map[0x25]='$';
398 shift_map[0x2E]='%';
399 shift_map[0x36]='^';
400 shift_map[0x3D]='&';
401 shift_map[0x3E]='*';
402 shift_map[0x46]='(';
403
404 shift_map[0x54]='{';
405 shift_map[0x0E]='~';
406 shift_map[0x4E]='_';
407 shift_map[0x55]='+';
408 shift_map[0x5D]='|';
409 shift_map[0x52]='"';
410 shift_map[0x49]='>';
411 shift_map[0x41]='<';
412 shift_map[0x4C]=': ';
413 shift_map[0x5B]='}';
414 shift_map[0x4A]='?';
415

```

```

416 shift_map[0x29]=0x20;//SPACE
417 shift_map[0x5A]='\n';//ENTER - set to newline for debugging purposes
418 shift_map[0x76]=0x1B;//ESC
419 shift_map[0x66]=0x08;//BKSP
420 shift_map[0x0D]=0x09;//TAB
421
422 shift_map[0x7C]='*';///'KP *';
423 shift_map[0x79]='+';///'KP +';
424 shift_map[0x7B]='-';///'KP -';
425 shift_map[0x71]='.';///'KP .';
426 shift_map[0x70]='0';///'KP 0';
427 shift_map[0x69]='1';///'KP 1';
428 shift_map[0x72]='2';///'KP 2';
429 shift_map[0x7A]='3';///'KP 3';
430 shift_map[0x6B]='4';///'KP 4';
431 shift_map[0x73]='5';///'KP 5';
432 shift_map[0x74]='6';///'KP 6';
433 shift_map[0x6C]='7';///'KP 7';
434 shift_map[0x75]='8';///'KP 8';
435 shift_map[0x7D]='9';///'KP 9';
436
437 ctl_map[0x1e] = 0x00; ///^@
438 ctl_map[0x36] = 0x1e; ///^^
439 ctl_map[0x3e] = 0x7f; ///^?
440 ctl_map[0x4e] = 0x1f; ///^_
441 ctl_map[0x15] = 0x11; ///^Q
442 ctl_map[0x1d] = 0x17; ///^W
443 ctl_map[0x24] = 0x05; ///^E
444 ctl_map[0x2d] = 0x12; ///^R
445 ctl_map[0x2c] = 0x14; ///^T
446 ctl_map[0x35] = 0x19; ///^Y
447 ctl_map[0x3c] = 0x15; ///^U
448 ctl_map[0x44] = 0x0f; ///^O
449 ctl_map[0x4d] = 0x10; ///^P
450 ctl_map[0x54] = 0x1b; ///^[
451 ctl_map[0x5b] = 0x1d; ///^]
452 ctl_map[0x5d] = 0x1c; ///^\\ ...
453 ctl_map[0x1c] = 0x01; ///^A
454 ctl_map[0x1b] = 0x13; ///^S
455 ctl_map[0x2b] = 0x06; ///^F
456 ctl_map[0x34] = 0x07; ///^G
457 ctl_map[0x33] = 0x08; ///^H
458 ctl_map[0x3b] = 0x0a; ///^J
459 ctl_map[0x42] = 0x0b; ///^K
460 ctl_map[0x4b] = 0x0c; ///^L
461 ctl_map[0x1a] = 0x1a; ///^Z
462 ctl_map[0x22] = 0x18; ///^X
463 ctl_map[0x21] = 0x03; ///^C
464 ctl_map[0x2a] = 0x16; ///^V
465 ctl_map[0x32] = 0x02; ///^B
466 ctl_map[0x31] = 0x0e; ///^N
467 ctl_map[0x3a] = 0x0d; ///^M
468 ctl_map[0x23] = 0x04; ///^D
469 ctl_map[0x43] = 0x09; ///^I
470
471 }
472 // dac.c
473
474 #include "stm32f10x.h"
475 #include "dac.h"
476
477
478 static int wave[40] = {
479     2047,2367,2679,2976,3250,3494,3703,3870,3993,4068,4094,4068,3993,3870,3703,3494,3250,2976,
480     2679,2367,2047,1726,1414,1117,843,599,390,223,100,25,0,25,100,223,390,599,843,1117,1414,1726
481 };
482 int beepCount = 0;
483
484
485 void DAC_beep(void)

```

```

486 {
487     beepCount = 100;
488 }
489
490
491 void Tim3_init()
492 {
493     RCC->APB1ENR |= RCC_APB1ENR_TIM3EN;
494     NVIC->ISER[0] = NVIC_ISER_SETENA_29;
495     NVIC->IP[7] = 0; // Highest priority!
496     TIM3->CR1 = 0x94; // Count down, restart automatically, only update on under/overflow
497     TIM3->DIER = 1; // enable interrupt on timer finish
498
499     TIM3->ARR = 0xFFFF; // 8000 is 1ms on 8Mhz clock
500
501     TIM3->CR1 |= 1; // enable timer.
502 }
503
504 void TIM3_IRQHandler()
505 // Interrupt on ISER[0]0x20000000
506 // Output new DAC value along the wave form
507 {
508     if (beepCount)
509     {
510         static int count = 0;
511         DAC->DHR12R2 = wave[count++];
512         //DAC->SWTRIGR = 2;
513         if (count == 40)
514             count = 0;
515         beepCount--;
516     }
517
518     // reset interrupt pending in NVIC
519     TIM3->SR &= 0xFFFFF0;
520     NVIC->ICPR[0] = NVIC_ICPR_CLRPEND_29;
521 }
522
523
524 void DAC_init()
525 {
526     // Enable gpio clock
527     RCC->APB1ENR |= RCC_APB1ENR_DACEN;
528     // Setup GPIOs
529     // DAC_OUT2 is PA5
530     RCC->APB2ENR |= RCC_APB2ENR_IOPAEN;
531     GPIOA->CRL = (GPIOA->CRL & 0xFF0FFFF) | 0x00B00000; // PC5 set to output AF push-pull
532
533     // Write configs
534     DAC->CR = 0x010000; // enable DAC channel 2, turn off buffering
535     //DAC->CR |= 0x3C0000; // enable triggers on software trigger
536
537     Tim3_init();
538 }
539
540
541 //
542 // lcd.c
543 // ECE 3710 Microcontroller H&S
544 // Utah State University
545 // Written by Kelly Hathaway
546 // And William Hatch and Scott Sorensen
547 //
548
549 #include "stm32f10x.h"
550 #include "lcd.h"
551 #include "ascii.h"
552
553 #define WR_low_Pin 0x0002 // Pin 1
554 #define RD_low_Pin 0x0004 // Pin 2
555 #define CS_low_Pin 0x0040 // Pin 6

```

```

556 #define DC_Pin      0x0080 // Pin 7
557
558
559 // configuration of the LCD port pins
560 void LCD_Config(void)
561 {
562     unsigned int config_temp;
563
564     RCC->APB2ENR |= 0x1D; // Enable port A, B, and C
565
566     config_temp = GPIOA->CRL; // Pin A.3 for Back light
567     config_temp &= ~0x0000F00;
568     config_temp |= 0x00003000;
569     GPIOA->CRL = config_temp;
570
571     GPIOB->CRL = 0x33333333; // Port B for Data[15:0] pins
572     GPIOB->CRH = 0x33333333;
573
574     config_temp = GPIOC->CRL; // PC.0(LCD RST), PC.1(WR), PC.2(RD) , PC.6(CS), PC.7(DC)
575     config_temp &= ~0xFF00FFF;
576     config_temp |= 0x33000333;
577     GPIOC->CRL = config_temp;
578 }
579
580 void LCD_Initialization(void)
581 {
582     unsigned int config_temp;
583
584     LCD_Config();
585
586     config_temp = AFIO->MAPR; // enable SW Disable JTAG
587     config_temp &= ~0x0700000;
588     config_temp |= 0x0200000;
589     AFIO->MAPR = config_temp;
590
591     GPIOC->BRR = 0x0001; // LCD reset
592     delay_ms(100);
593     GPIOC->BSRR = 0x0001;
594     GPIOA->BSRR = 0x0008; // back light
595
596     LCD_WriteReg(0x0000,0x0001); delay_ms(50); /* Enable LCD Oscillator */
597     LCD_WriteReg(0x0003,0xA8A4); delay_ms(50); // Power control(1)
598     LCD_WriteReg(0x000C,0x0000); delay_ms(50); // Power control(2)
599     LCD_WriteReg(0x000D,0x080C); delay_ms(50); // Power control(3)
600     LCD_WriteReg(0x000E,0x2B00); delay_ms(50); // Power control(4)
601     LCD_WriteReg(0x001E,0x00B0); delay_ms(50); // Power control(5)
602     LCD_WriteReg(0x0001,0x2B3F); delay_ms(50); // Driver Output Control /* 320*240 0x2B3F */
603     LCD_WriteReg(0x0002,0x0600); delay_ms(50); // LCD Drive AC Control
604     LCD_WriteReg(0x0010,0x0000); delay_ms(50); // Sleep Mode off
605     // LCD_WriteReg(0x0011,0x6070); delay_ms(50); // Entry Mode ## flip bit
606     LCD_WriteReg(0x0011,0x6078); delay_ms(50); // Entry Mode
607     ## flip bit 3 to switch horiz/vert auto-update on write
608     LCD_WriteReg(0x0005,0x0000); delay_ms(50); // Compare register(1)
609     LCD_WriteReg(0x0006,0x0000); delay_ms(50); // Compare register(2)
610     LCD_WriteReg(0x0016,0xEF1C); delay_ms(50); // Horizontal Porch
611     LCD_WriteReg(0x0017,0x0003); delay_ms(50); // Vertical Porch
612     LCD_WriteReg(0x0007,0x0133); delay_ms(50); // Display Control
613     LCD_WriteReg(0x000B,0x0000); delay_ms(50); // Frame Cycle control
614     LCD_WriteReg(0x000F,0x0000); delay_ms(50); // Gate scan start position
615     LCD_WriteReg(0x0041,0x0000); delay_ms(50); // Vertical scroll control(1)
616     LCD_WriteReg(0x0042,0x0000); delay_ms(50); // Vertical scroll control(2)
617     LCD_WriteReg(0x0048,0x0000); delay_ms(50); // First window start
618     LCD_WriteReg(0x0049,0x013F); delay_ms(50); // First window end
619     LCD_WriteReg(0x004A,0x0000); delay_ms(50); // Second window start
620     LCD_WriteReg(0x004B,0x0000); delay_ms(50); // Second window end
621     LCD_WriteReg(0x0044,0xEF00); delay_ms(50); // Horizontal RAM address position
622     LCD_WriteReg(0x0045,0x0000); delay_ms(50); // Vertical RAM address start position
623     LCD_WriteReg(0x0046,0x013F); delay_ms(50); // Vertical RAM address end position
624     LCD_WriteReg(0x0030,0x0707); delay_ms(50); // gamma control(1)

```

```

624     LCD_WriteReg(0x0031,0x0204);    delay_ms(50);    // gamma control(2)
625     LCD_WriteReg(0x0032,0x0204);    delay_ms(50);    // gamma control(3)
626     LCD_WriteReg(0x0033,0x0502);    delay_ms(50);    // gamma control(4)
627     LCD_WriteReg(0x0034,0x0507);    delay_ms(50);    // gamma control(5)
628     LCD_WriteReg(0x0035,0x0204);    delay_ms(50);    // gamma control(6)
629     LCD_WriteReg(0x0036,0x0204);    delay_ms(50);    // gamma control(7)
630     LCD_WriteReg(0x0037,0x0502);    delay_ms(50);    // gamma control(8)
631     LCD_WriteReg(0x003A,0x0302);    delay_ms(50);    // gamma control(9)
632     LCD_WriteReg(0x003B,0x0302);    delay_ms(50);    // gamma control(10)
633     LCD_WriteReg(0x0023,0x0000);    delay_ms(50);    // RAM write data mask(1)
634     LCD_WriteReg(0x0024,0x0000);    delay_ms(50);    // RAM write data mask(2)
635     LCD_WriteReg(0x0025,0x8000);    delay_ms(50);    // Frame Frequency
636     LCD_WriteReg(0x004f,0);        // Set GDDRAM Y address counter
637     LCD_WriteReg(0x004e,0);        // Set GDDRAM X address counter
638
639     delay_ms(50);
640 }
641
642 // Paints the LCD with Color
643 void LCD_Clear( unsigned short Color )
644 {
645     unsigned int i;
646
647     LCD_SetCursor(0,0);
648
649     GPIOC->BRR = CS_low_Pin;
650
651     LCD_WriteIndex( 0x0022 );
652     for( i=0; i< MAX_X*MAX_Y; i++ )
653         LCD_WriteData( Color );
654
655     GPIOC->BSRR = CS_low_Pin;
656 }
657
658 // Write a command
659 void LCD_WriteIndex( unsigned short index )
660 {
661     LCD_Write_Generic(index, 0);
662 }
663
664 // Write data
665 void LCD_WriteData( unsigned short data )
666 {
667     LCD_Write_Generic(data, 1);
668 }
669
670 // Write generic...
671 void LCD_Write_Generic(unsigned short toWrite, unsigned short dataBool)
672 {
673     unsigned short pc_ops = GPIOC->ODR;
674
675     // Configure Ports - done in LCD init function
676     // Set control bits (RD, WR, D/C, CS)
677     // 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)
678
679     pc_ops &= 0xFF38; // unset RST,WR,RD,CS,DC
680     pc_ops |= 0x0007; // set RD = 1, and WR = 1, and RST = 1
681     if (dataBool) pc_ops |= 0x80; // set DC = 1 if we want to write data
682
683     GPIOC->ODR = pc_ops;
684     delay_ms(0);
685
686     // Write data bits
687     GPIOB->ODR = toWrite;
688     delay_ms(0);
689
690     GPIOC->ODR = (pc_ops & 0xFFFFD); // unset WR
691     GPIOC->ODR = pc_ops; // set WR
692

```

```

693 }
694
695 //
696 void LCD_WriteReg( unsigned short LCD_Reg, unsigned short LCD_RegValue )
697 {
698     GPIOC->BRR = CS_low_Pin;
699
700     LCD_WriteIndex( LCD_Reg );
701     LCD_WriteData( LCD_RegValue );
702
703     GPIOC->BSRR = CS_low_Pin;
704 }
705
706 // Set cursor to x y address
707 void LCD_SetCursor( unsigned short x, unsigned int y )
708 {
709     #if ( DISP_ORIENTATION == 90 ) || ( DISP_ORIENTATION == 270 )
710         unsigned short swap_temp;
711
712         y = (MAX_Y-1) - y;
713         swap_temp = y;
714         y = x;
715         x = swap_temp;
716
717     #elif ( DISP_ORIENTATION == 0 ) || ( DISP_ORIENTATION == 180 )
718         y = (MAX_Y-1) - y;
719
720     #endif
721
722     LCD_WriteReg( 0x004E, x );
723     LCD_WriteReg( 0x004F, y );
724 }
725
726 void delay_ms( unsigned int ms )
727 {
728     int i;
729     while(ms--)
730     {
731         //for(i = 0; i < 1669; ++i); // 1 ms delay loop
732         for(i = 0; i < 8676; ++i); // 1 ms delay loop
733     }
734 }
735
736 void LCD_DrawSquareY( unsigned short x, unsigned short y, unsigned short w, unsigned short h,
737 unsigned short color )
738 {
739     unsigned int i,j;
740
741     LCD_SetCursor(x,y);
742
743     GPIOC->BRR = CS_low_Pin;
744
745     for (j=0; j < w; j++ )
746     {
747         LCD_SetCursor(x+j, y);
748         LCD_WriteIndex( 0x0022 );
749         for( i=0; i < h; i++ )
750             LCD_WriteData( color );
751     }
752
753     GPIOC->BSRR = CS_low_Pin;
754 }
755
756 void LCD_DrawSquare( unsigned short x, unsigned short y, unsigned short w, unsigned short h,
757 unsigned short color )
758 {
759     unsigned int i,j;
760

```



```

761     LCD_SetCursor(x,y);
762
763     GPIOC->BRR = CS_low_Pin;
764
765
766         for (j=0; j < w; j++ )
767         {
768             LCD_SetCursor(x, y+j);
769             LCD_WriteIndex( 0x0022 );
770             for( i=0; i < h; i++ )
771                 LCD_WriteData( color );
772         }
773
774     GPIOC->BSRR = CS_low_Pin;
775 }
776
777 void LCD_DrawCharacterOnY (unsigned short x, unsigned short y, unsigned short fgColor, unsigned
778 short bgColor, unsigned char symbol)
779 // Draws a character oriented so that left to right goes along the positive y axis
780 {
781     unsigned char ascii_buf[16];
782     unsigned char line;
783     int i, j;
784
785     LCD_SetCursor(x,y);
786     GPIOC->BRR = CS_low_Pin;
787
788     get_ascii(ascii_buf, symbol);
789
790     for (i = 0; i < 16; ++i)
791     {
792         line = ascii_buf[i];
793         LCD_SetCursor(x+i, y);
794         LCD_WriteIndex( 0x0022 );
795         for (j = 0; j < 8; ++j)
796         {
797             if (line & (0x80 >> j))
798                 LCD_WriteData( fgColor );
799             else
800                 LCD_WriteData( bgColor );
801             //delay_ms(1);
802         }
803     }
804     GPIOC->BSRR = CS_low_Pin;
805 }
806 void LCD_DrawCharacterOnX (unsigned short x, unsigned short y, unsigned short fgColor, unsigned
807 short bgColor, unsigned char symbol, unsigned char underline)
808 // Draws a character oriented so that left to right goes along the positive X axis
809 {
810     unsigned char ascii_buf[16];
811     unsigned char line;
812     int i, j;
813
814     LCD_SetCursor(x,y);
815     GPIOC->BRR = CS_low_Pin;
816
817     get_ascii(ascii_buf, symbol);
818
819     for (i = 0; i < 16; ++i)
820     {
821         line = ascii_buf[i];
822         if (i == 15 && underline)
823             line = 0xFF;
824         LCD_SetCursor(x, y+i);
825         LCD_WriteIndex( 0x0022 );
826         for (j = 0; j < 8; ++j)
827         {
828             if (line & (0x80 >> j))

```

```

829         else
830             LCD_WriteData( bgColor );
831         //delay_ms(1);
832     }
833 }
834 GPIOC->BSRR = CS_low_Pin;
835 }
836 void LCD_DrawChar_rc (unsigned int row, unsigned int col, unsigned short fgColor, unsigned short
bgColor, unsigned char symbol, unsigned char underline)
837 // Draws a character on the givel row and column
838 {
839     //LCD_DrawCharacterOnY(row * 16, MAX_Y - 12 - (col*8), fgColor, bgColor, symbol);
840     LCD_DrawCharacterOnX(col*8, row*16, fgColor, bgColor, symbol, underline);
841 }
842
843 void LCD_WriteCharactersOnY (unsigned short x, unsigned short y, unsigned short fgColor, unsigned
short bgColor, char* words, int maxLength)
844 // Draws a line of characters increasing on Y axis
845 {
846     int i;
847     for (i = 0; i < maxLength; ++i)
848     {
849         if(words[i] == 0)
850             break;
851         LCD_DrawCharacterOnY(x, y - (8*i), fgColor, bgColor, words[i]);
852     }
853 }
854 void LCD_WriteLinesOnY(unsigned short lineNumber, unsigned short fgColor, unsigned short bgColor,
char* words, char drawToLineEnd)
855 {
856     int len, numLines, lastLineLength, i, curLine;
857     char spaces[CHARS_HORIZ_ON_Y];
858     len = strlen(words);
859     numLines = len / CHARS_HORIZ_ON_Y;
860     lastLineLength = len % CHARS_HORIZ_ON_Y;
861     if(lastLineLength != 0)
862         numLines++;
863
864     for(curLine = 0; curLine < numLines; ++curLine)
865     {
866         LCD_WriteCharactersOnY((lineNumber + curLine) * 16, MAX_Y-12, fgColor, bgColor,
words+((CHARS_HORIZ_ON_Y) * curLine), CHARS_HORIZ_ON_Y);
867     }
868     // TODO: fix draw to end of last line
869     if(lastLineLength != 0)
870     {
871         for(i = 0; i < CHARS_HORIZ_ON_Y; ++i)
872         {
873             spaces[i] = ' ';
874         }
875         LCD_WriteCharactersOnY((lineNumber + curLine - 1) * 16, MAX_Y-12-
(lastLineLength*8), fgColor, bgColor, spaces, CHARS_HORIZ_ON_Y-lastLineLength);
876     }
877 }
878
879
880
881
882 // END OF FILE
883 // main.c
884 // ECE 3710
885 // Final Project
886 // By William Hatch and Scott Sorensen
887
888
889 #include "stm32f10x.h"
890 #include "lcd.h"
891 #include "usart2.h"
892 #include "ps2_over_gpioc.h"
893 #include "code_to_ascii.h"

```

```

894 #include "terminal.h"
895 #include "dac.h"
896
897
898
899 void SystemInit(void)
900 {
901 }
902
903 void handlePs2Data(void);
904 void handleUsartData(void);
905
906
907
908 extern int cursorCol, cursorRow;
909 extern char screenChars[ROWS][COLS];
910 extern unsigned short screenFgColor[ROWS][COLS];
911 extern unsigned short screenBgColor[ROWS][COLS];
912
913
914
915 int main()
916 {
917     // Enable external oscillator
918     //RCC->CFGR = 0x0418000A; // Mult PLL by 8 = 32 MHz
919     RCC->CFGR = 0x0428000A; // 48MHz
920     /// 418->428 makes it 48Mhz
921     RCC->CR = 0x03004583; //USE PLL Clock for SW and MC
922
923
924     // Initialize everything
925     LCD_Initialization();
926     usart2_init();
927     ps2_over_gpioc_init();
928
929     LCD_Clear(Black);
930     scan_code_init();
931     DAC_init();
932     bufClear();
933
934     while(1)
935     {
936         handlePs2Data();
937         handleUsartData();
938         flushScreen();
939     }
940 }
941
942
943 void handlePs2Data()
944 {
945     int bytes;
946     int i;
947     unsigned char buf[PS2_DATA_SIZE];
948
949     // copy buffer
950     bytes = ps2_memcpy(buf);
951
952     // tx buffer
953     for (i = 0; i < bytes; ++i)
954     {
955         usart2_tx(buf[i]);
956     }
957 }
958
959 void handleUsartData()
960 {
961     int bytes;
962     unsigned char buf[USART2_DATA_SIZE];
963     // copy buffer

```

```

964         bytes = usart2_memcpy(buf);
965
966         // push buffered data to display
967         handleAscii(buf, bytes);
968     }
969
970
971     void HardFault_Handler()
972     {
973         LCD_Clear(Red);
974         //Reset_Handler();
975     }
976
977 //ps2_over_gpioc.c
978
979 #include "stm32f10x.h"
980 #include "ps2_over_gpioc.h"
981 #include "usart2.h"
982 #include "code_to_ascii.h"
983 #include "lcd.h"
984
985 unsigned char ps2_data[PS2_DATA_SIZE];
986 int ps2_bytes_rec = 0;
987
988 extern unsigned char map[0x100];
989 extern unsigned char shift_map[0x100];
990 extern unsigned char ctl_map[0x100];
991
992 void ps2_insert_to_buffer(char *insert, int size)
993 {
994     int i;
995
996     if (ps2_bytes_rec + size > PS2_DATA_SIZE)
997         return;
998
999     NVIC->ICER[0] = NVIC_ICER_CLRENA_10;
1000     for (i = 0; i < size; ++i)
1001     {
1002         ps2_data[ps2_bytes_rec++] = insert[i];
1003     }
1004     NVIC->ISER[0] = NVIC_ISER_SETENA_10;
1005 }
1006
1007 // Switch PC0 to PC4, and PC1 to PC3
1008 void ps2_over_gpioc_init(void)
1009 {
1010     // Setup EXTI0
1011     EXTI->IMR |= EXTI_IMR_MR4;
1012     //EXTI->EMR |= EXTI_EMR_MR4;
1013     EXTI->RTSR |= EXTI_RTSR_TR4;
1014
1015     RCC->APB2ENR |= RCC_APB2ENR_IOPCEN;
1016
1017     // Setup AFIO events
1018     RCC->APB2ENR |= 1; // enable AFIO clock
1019     //AFIO->EVCR = 0b10100000; // Events enabled on PC0
1020     //AFIO->EVCR = AFIO_EVCR_EVOE | AFIO_EVCR_PORT_PC | AFIO_EVCR_PIN_PX0;
1021     AFIO->EXTICR[1] = (AFIO->EXTICR[2] & 0xFFFFFFF0) | 0x2;
1022     GPIOC->CRL = (GPIOC->CRL & 0xFFFF0FFF) | 0x88000; // Configure PC[3-4] for pull-up/pull-
down input
1023     GPIOC->ODR |= 0x18; // set bits of ODR so we can read the input.
1024
1025     // Setup NVIC
1026     // EXT4 is interrupt number 10
1027     NVIC->ISER[0] = NVIC_ISER_SETENA_10;
1028
1029     // Set priority - this should be our high priority interrupt
1030     //NVIC->IP[1] = (NVIC->IP[1] & 0xFF00FFFF) | 0x00FF0000;
1031
1032 }

```

```

1033
1034 void EXTI4_IRQHandler(void)
1035 {
1036     static unsigned int calls = 0;
1037     static unsigned char rx_state = 0; // 0 not receiving, 1 receiving
1038     static unsigned char data_bits_rec = 0;
1039     static unsigned char data = 0;
1040     static unsigned char stop_bit_rec = 0;
1041     static unsigned char last_byte_was_escape = 0;
1042     static unsigned char shift_on = 0;
1043     static unsigned char ctl_on = 0;
1044     static unsigned char alt_on = 0;
1045
1046     unsigned char bit, ascii;
1047
1048     calls++;
1049     bit = GPIOC->IDR & 0x8; // read pc3 (data line)
1050     bit = bit << 4;
1051     while(1)
1052     {
1053         // Handle start bit
1054         if (!rx_state)
1055         {
1056             if(bit)
1057                 break;
1058             rx_state = 1;
1059             data_bits_rec = 0;
1060             data = 0;
1061             stop_bit_rec = 0;
1062
1063             break;
1064         }
1065
1066         // Receive data
1067         if (data_bits_rec < 8)
1068         {
1069             data = data >> 1;
1070             data |= bit;
1071             data_bits_rec++;
1072             break;
1073         }
1074
1075         // Handle stop/acknowledge bits
1076         if (!stop_bit_rec)
1077         {
1078             stop_bit_rec = 1;
1079
1080
1081             // Map scancodes to ascii, throwing away everything but lowercase
1082             // alphanumeric characters and spaces.
1083
1084             if (last_byte_was_escape)
1085             {
1086                 last_byte_was_escape = 0;
1087                 if(shift_on && (data == 0x12 | data == 0x59))
1088                     shift_on = 0;
1089                 if(ctl_on && (data == 0x14))
1090                     ctl_on = 0;
1091                 if(alt_on && (data == 0x11))
1092                     alt_on = 0;
1093                 break;
1094             }
1095             if (data == 0xF0) // key up escape
1096             {
1097                 last_byte_was_escape = 1;
1098                 break;
1099             }
1100
1101             if (data == 0x12 | data == 0x59) // key down shift
1102             {

```

```

1103         shift_on = 1;
1104         break;
1105     }
1106     if (data == 0x14) // key down ctl
1107     {
1108         ctl_on = 1;
1109         break;
1110     }
1111     if (data == 0x11) // key down alt
1112     {
1113         alt_on = 1;
1114         break;
1115     }
1116
1117     if (ctl_on)
1118         ascii = ctl_map[data];
1119     else if (shift_on)
1120         ascii = shift_map[data];
1121     else
1122         ascii = map[data];
1123
1124
1125
1126
1127     // put data into buffer
1128     if (ps2_bytes_rec < PS2_DATA_SIZE-1)
1129     {
1130         if(alt_on)
1131         {
1132             ps2_data[ps2_bytes_rec] = 0x1b; // escape
1133             ps2_bytes_rec++;
1134         }
1135         ps2_data[ps2_bytes_rec] = ascii;
1136         //ps2_data[ps2_bytes_rec] = data;
1137         ps2_bytes_rec++;
1138     }
1139
1140
1141     break;
1142 }
1143
1144
1145 // Here we will let one clock cycle pass, because there may
1146 // be an acknowledgement bit, but we're not sure how to handle it.
1147
1148 rx_state = 0;
1149 break;
1150 }
1151
1152 //NVIC->ICPR[0] = 0x40;
1153 EXTI->PR = 0x10; // Clear EXTI pending
1154 NVIC->ICPR[0] = NVIC_ICPR_CLRPEND_10;
1155 return;
1156 }
1157
1158 void ps2_dump_data_over_usart2()
1159 // Dumps data in ps2_data over usart2
1160 {
1161     int i;
1162
1163     for (i = 0; i < ps2_bytes_rec; ++i)
1164     {
1165         usart2_tx(ps2_data[i]);
1166         LCD_DrawCharacterOnY(40,40,Blue, Black, ps2_data[i]);
1167     }
1168     ps2_bytes_rec =0;
1169 }
1170
1171 int ps2_memcpy(unsigned char * dst)
1172 {

```

```

1173     int ret, i;
1174     // turn off interrupts
1175     NVIC->ICER[0] = NVIC_ICER_CLRENA_10;
1176     // copy buffer
1177     for(i = 0; i < ps2_bytes_rec; ++i)
1178     {
1179         dst[i] = ps2_data[i];
1180     }
1181     ret = ps2_bytes_rec;
1182
1183     ps2_bytes_rec = 0;
1184     // enable interrupts
1185     NVIC->ISER[0] = NVIC_ISER_SETENA_10;
1186     return ret;
1187 }
1188
1189
1190
1191 // terminal.c
1192
1193 #include "terminal.h"
1194 #include "lcd.h"
1195 #include "ps2_over_gpioc.h"
1196 #include "dac.h"
1197
1198 #define CURSOR_STACK_SIZE 200
1199 int cursorStack[2][2][CURSOR_STACK_SIZE]; // one for CSI stack, one for ESC (non-CSI stack)
1200 int CSICursorStackPointer=-1;
1201 int ESCcursorStackPointer=-1;
1202 int cursorCol = 0; // cursor column number
1203 int cursorRow = 0; // cursor line number
1204
1205 unsigned short currFgColor = TermDefault;
1206 unsigned short currBgColor = TermBlack;
1207
1208 unsigned char currDisplayOps = 0; // for flags for underline, blink, etc
1209
1210 char screenChars[ROWS][COLS];
1211 char oldScreenChars[ROWS][COLS];
1212 unsigned short screenFgColor[ROWS][COLS];
1213 unsigned short screenBgColor[ROWS][COLS];
1214 unsigned short oldScreenFgColor[ROWS][COLS];
1215 unsigned short oldScreenBgColor[ROWS][COLS];
1216 int screenTop = 0;
1217 int oldScreenTop = 0;
1218
1219 unsigned char screenDisplayOps[ROWS][COLS];
1220 unsigned char oldScreenDisplayOps[ROWS][COLS];
1221 unsigned char drawCursor = 1;
1222 #define CURSOR_CHAR 1 // attribute for character that the cursor is on
1223 #define UNDERLINE 2
1224 #define BOLD 4
1225 #define BLINK 8
1226 #define REVERSE_VIDEO 16
1227 #define HALF_BRIGHT 32
1228
1229
1230
1231 ////////////////////////////////////////////////// Ascii Control Functions //////////////////////////////////////////
1232
1233 void addCursorAttr()
1234 { // add cursor attribute to screenChar at current cursor location
1235     int rowTr; // translated row
1236     rowTr = (cursorRow + screenTop) % ROWS;
1237     screenDisplayOps[rowTr][cursorCol] |= CURSOR_CHAR;
1238 }
1239 void remCursorAttr()
1240 { // remove cursor attribute to screenChar at current cursor location
1241     int rowTr; // translated row
1242     rowTr = (cursorRow + screenTop) % ROWS;

```

```

1243     screenDisplayOps[rowTr][cursorCol] &= ~CURSOR_CHAR;
1244 }
1245 void clearExtraneousCursors()
1246 { // since we have extra cursors, let's add this cludge to clear them.
1247     int i,j;
1248     for(i = 0; i < ROWS; ++i)
1249     {
1250         for(j = 0; j < COLS; ++j)
1251         {
1252             screenDisplayOps[i][j] &= ~CURSOR_CHAR;
1253         }
1254     }
1255     addCursorAttr();
1256 }
1257
1258 void do_LF()
1259 // Do a standard line feed
1260 {
1261     int i, rowTr;
1262
1263     remCursorAttr();
1264     if (cursorRow < ROWS - 1)
1265         cursorRow = (cursorRow + 1);
1266     else
1267     {
1268         screenTop = (screenTop+1) % ROWS;
1269         rowTr = (ROWS - 1 + screenTop) % ROWS;
1270         for(i = 0; i < COLS; ++i)
1271         {
1272             screenChars[rowTr][i] = ' ';
1273         }
1274     }
1275     addCursorAttr();
1276 }
1277
1278 void do_CR()
1279 // Do a standard carriage return
1280 {
1281     remCursorAttr();
1282     cursorCol = 0;
1283     addCursorAttr();
1284 }
1285
1286 void handle_LF()
1287 // Handle the \n character
1288 // In newline mode this gives both LF and CR
1289 {
1290     do_LF();
1291     // TODO - check somehow whether newline mode is on (it probably always will be
1292     // if (newLineMode)
1293     do_CR();
1294 }
1295
1296 void handle_CR()
1297 // handle CR character
1298 {
1299     // I think this may be the place to do this line blackout...
1300     int j;
1301     for(j = cursorCol; j < COLS; ++j)
1302     {
1303         screenChars[(cursorRow + screenTop)%ROWS][j] = ' ';
1304     }
1305     do_CR();
1306 }
1307
1308 void advance_cursor()
1309 // advances the cursor, duh.
1310 {
1311     int i;
1312     remCursorAttr();

```



```

1313         if(cursorCol >= COLS - 1)
1314         { // It seems many terminals don't automatically scroll, and it breaks some
functionality.
1315             // Unless I make a termcap entry for this terminal or figure out how it works,
it breaks functionality of
1316             // programs like vim in screen... odd...
1317             cursorCol = 0;
1318             if (cursorRow < ROWS - 1)
1319                 cursorRow = (cursorRow + 1);
1320             else
1321             { // Scroll screen
1322                 screenTop = (screenTop + 1) % ROWS;
1323                 for (i = 0; i < COLS; ++i)
1324                 { // clear the new bottom line to be blank as it scrolls in.
1325                     screenChars[(screenTop+ROWS-1)%ROWS][i] = ' ';
1326                     screenDisplayOps[(screenTop+ROWS-1)%ROWS][i] = 0;
1327                     screenFgColor[(screenTop+ROWS-1)%ROWS][i] = TermDefault;
1328                     screenBgColor[(screenTop+ROWS-1)%ROWS][i] = TermBlack;
1329                 }
1330             }
1331         }
1332         else
1333             cursorCol++;
1334         addCursorAttr();
1335     }
1336
1337 void handle_normal(char ascii)
1338 // handles normal characters for printing to the screen
1339 {
1340     int rowTr; // translated row
1341     rowTr = (cursorRow + screenTop) % ROWS;
1342     screenChars[rowTr][cursorCol] = ascii;
1343     screenFgColor[rowTr][cursorCol] = currFgColor;
1344     screenBgColor[rowTr][cursorCol] = currBgColor;
1345     screenDisplayOps[rowTr][cursorCol] = currDisplayOps;
1346     advance_cursor();
1347 }
1348
1349 void moveCursor(unsigned int num, unsigned int dir)
1350 {
1351     #define UP 1
1352     #define DOWN 2
1353     #define LEFT 3
1354     #define RIGHT 4
1355     remCursorAttr();
1356     if (dir == UP)
1357     {
1358         cursorRow -= num;
1359         if (cursorRow < 0)
1360             cursorRow = 0;
1361     }
1362     else if (dir == DOWN)
1363     {
1364         cursorRow += num;
1365         if (cursorRow > ROWS - 1)
1366             cursorRow = ROWS - 1;
1367     }
1368     else if (dir == RIGHT)
1369     {
1370         cursorCol += num;
1371         if (cursorCol > COLS - 1)
1372             cursorCol = COLS - 1;
1373     }
1374     else if (dir == LEFT)
1375     {
1376         cursorCol -= num;
1377         if (cursorCol < 0)
1378             cursorCol = 0;
1379     }
1380     addCursorAttr();

```

```

1381 }
1382
1383 void handle_colorCodes(unsigned int *csi_numbers, unsigned int csi_numbers_rec)
1384 {
1385     int i;
1386     for (i = 0; i < csi_numbers_rec; ++i)
1387     {
1388         switch (csi_numbers[i])
1389         {
1390             case 0:
1391                 currBgColor = TermBlack;
1392                 currFgColor = TermDefault;
1393                 currDisplayOps = 0;
1394                 break;
1395             case 1:
1396                 currDisplayOps |= BOLD;
1397                 break;
1398             case 2:
1399                 // set half-bright
1400                 currDisplayOps |= HALF_BRIGHT;
1401                 break;
1402             case 4:
1403                 // set underscore
1404                 currDisplayOps |= UNDERLINE;
1405                 break;
1406             case 5:
1407                 // set blink
1408                 currDisplayOps |= BLINK;
1409                 break;
1410             case 7:
1411                 // set reverse video (whatever that is)
1412                 currDisplayOps |= REVERSE_VIDEO;
1413                 break;
1414             case 21:
1415                 // set normal intensity (bold off?)
1416                 currDisplayOps &= ~BOLD;
1417                 break;
1418             case 22:
1419                 // set normal intensity (half-bright off?)
1420                 currDisplayOps &= ~HALF_BRIGHT;
1421                 break;
1422             case 24:
1423                 // set underline off
1424                 currDisplayOps &= ~UNDERLINE;
1425                 break;
1426             case 25:
1427                 // set blink off
1428                 currDisplayOps &= ~BLINK;
1429                 break;
1430             case 27:
1431                 // set reverse video off
1432                 currDisplayOps &= ~REVERSE_VIDEO;
1433                 break;
1434
1435             // 30-49 are mostly basic color options
1436             case 30:
1437                 currFgColor = TermBlack;
1438                 if(currDisplayOps & BOLD)
1439                     currFgColor = TermBlackBright;
1440                 break;
1441             case 31:
1442                 currFgColor = TermRed;
1443                 if(currDisplayOps & BOLD)
1444                     currFgColor = TermRedBright;
1445                 break;
1446             case 32:
1447                 currFgColor = TermGreen;
1448                 if(currDisplayOps & BOLD)
1449                     currFgColor = TermGreenBright;
1450                 break;

```

```

1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520

    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;
}
}
}

```

```

1521
1522
1523 void keepCursorInBounds()
1524 {
1525     if (cursorRow >= ROWS)
1526     {
1527         cursorRow = ROWS-1;
1528     }
1529     else if (cursorRow < 0)
1530         cursorRow = 0;
1531     if (cursorCol >= COLS-1)
1532         cursorCol = COLS-1;
1533     else if (cursorCol < 0)
1534         cursorCol = 0;
1535 }
1536
1537 void reportCursorToHost()
1538 {
1539     char rowColResponse[10];
1540     char rowNum[3];
1541     char colNum[3];
1542     int divisor, i,j;
1543     char firstSeen = 0;
1544     int rowsSent=0, colsSent=0;
1545     // echo ESC[<ROW>;<COL>R
1546
1547     for (i = 0, divisor = 100; divisor; ++i, divisor/=10)
1548     {
1549         rowNum[i] = ((cursorRow+1)/divisor) % 10;
1550         colNum[i] = ((cursorCol+1)/divisor) % 10;
1551     }
1552     rowColResponse[0] = 033; // ESC
1553     rowColResponse[1]='[';
1554     for(i = 2, j = 0; j < 4; ++j) // i is response position, j is loop position
1555     {
1556         if(rowNum[j] || j == 3) // only start putting out characters if we've seen the
first character or it's all 0
1557             firstSeen = 1;
1558         if(firstSeen)
1559         {
1560             rowColResponse[i] = rowNum[j];
1561             ++i;
1562             ++rowsSent;
1563         }
1564     }
1565     rowColResponse[i++] = ';';
1566     firstSeen = 0;
1567     for(j = 0; j < 4; ++j) // i is response position and keeps its previous value, j is loop
position
1568     {
1569         if(colNum[j])
1570             firstSeen = 1;
1571         if(firstSeen)
1572         {
1573             rowColResponse[i] = colNum[j];
1574             ++i;
1575             ++colsSent;
1576         }
1577     }
1578     rowColResponse[i] = 'R';
1579
1580     ps2_insert_to_buffer(rowColResponse, rowsSent+colsSent+4);
1581 }
1582
1583 void CSIpushCursor()
1584 {
1585     if(CSIcursorStackPointer < CURSOR_STACK_SIZE-1)
1586     {
1587         CSIcursorStackPointer++;
1588         cursorStack[0][0][CSIcursorStackPointer] = cursorRow;

```

```

1589         cursorStack[0][1][CSICursorStackPointer] = cursorCol;
1590     }
1591 }
1592 void CSIPopCursor()
1593 {
1594     remCursorAttr();
1595     if(CSICursorStackPointer >= 0)
1596     {
1597         cursorRow = cursorStack[0][0][CSICursorStackPointer];
1598         cursorCol = cursorStack[0][1][CSICursorStackPointer];
1599         CSICursorStackPointer--;
1600     }
1601     addCursorAttr();
1602 }
1603 void ESCpushCursor()
1604 {
1605     if(ESCcursorStackPointer < CURSOR_STACK_SIZE-1)
1606     {
1607         ESCcursorStackPointer++;
1608         cursorStack[0][0][ESCcursorStackPointer] = cursorRow;
1609         cursorStack[0][1][ESCcursorStackPointer] = cursorCol;
1610     }
1611 }
1612 void ESCpopCursor()
1613 {
1614     remCursorAttr();
1615     if(ESCcursorStackPointer >= 0)
1616     {
1617         cursorRow = cursorStack[0][0][ESCcursorStackPointer];
1618         cursorCol = cursorStack[0][1][ESCcursorStackPointer];
1619         ESCcursorStackPointer--;
1620     }
1621     addCursorAttr();
1622 }
1623
1624 ////////////////////////////////////////////////// Screen Manipulation ///////////////////////////////////
1625
1626 void bufClear(void)
1627 {
1628     int i,j;
1629     for(i = 0; i < ROWS; ++i)
1630     {
1631         for(j = 0; j < COLS; ++j)
1632         {
1633             screenChars[i][j] = ' ';
1634             screenFgColor[i][j] = Grey;
1635             screenBgColor[i][j] = Black;
1636             screenDisplayOps[i][j] = 0;
1637         }
1638     }
1639 }
1640
1641
1642 void flushScreen()
1643 {
1644     int i,j;
1645     int Io, In; // Translated i for screen top differences due to scrolling
1646     char letter;
1647     static int count = 0;
1648     static unsigned int refreshes = 0;
1649     #define REFRESH_COUNT 20
1650     clearExtraneousCursors();
1651     count = (count+1) % (REFRESH_COUNT +1);
1652     if (count == REFRESH_COUNT)
1653         refreshes++; // count number of refreshes, for blinking
1654
1655     for(i = ROWS-1; i >= 0; --i) // Start from the bottom of the screen
1656     {
1657         Io = (i + oldScreenTop) % ROWS;
1658         In = (i + screenTop) % ROWS;

```

```

1659         for(j = 0; j < COLS; ++j)
1660         {
1661             if (oldScreenChars[Io][j] != screenChars[In][j]
1662                 || oldScreenFgColor[Io][j] != screenFgColor[In][j]
1663                 || oldScreenBgColor[Io][j] != screenBgColor[In][j]
1664                 || oldScreenDisplayOps[Io][j] != screenDisplayOps[In][j]
1665                 || count == REFRESH_COUNT) // redraw everything every so often to
clear out glitches
1666             {
1667                 letter = screenChars[In][j];
1668                 if(screenDisplayOps[In][j] & BLINK && refreshes%2)
1669                 { // blink every other whole screen refresh cycle
1670                     letter = ' ';
1671                 }
1672                 if ((screenDisplayOps[In][j] & CURSOR_CHAR) && drawCursor ||
screenDisplayOps[In][j] & REVERSE_VIDEO)
1673                 {
1674                     LCD_DrawChar_rc (i, j, screenBgColor[In][j], screenFgColor
[In][j], letter, screenDisplayOps[In][j] & UNDERLINE);
1675                 }
1676                 else
1677                     LCD_DrawChar_rc (i, j, screenFgColor[In][j], screenBgColor
[In][j], letter, screenDisplayOps[In][j] & UNDERLINE);
1678             }
1679         }
1680     }
1681     for(i = 0; i < ROWS; ++i)
1682     {
1683         for (j = 0; j < COLS; ++j)
1684         {
1685             oldScreenChars[i][j] = screenChars[i][j];
1686             oldScreenFgColor[i][j] = screenFgColor[i][j];
1687             oldScreenBgColor[i][j] = screenBgColor[i][j];
1688         }
1689     }
1690     oldScreenTop = screenTop;
1691 }
1692
1693 ////////////////////////////////////////////////// Main Handler //////////////////////////////////////
1694
1695 void handleAscii(unsigned char *buf, int bytes)
1696 // Outputs ascii to screen or handles escape codes
1697 // Much of the escape code handling is in "man console_codes" in in the Linux Programmer's Manual
1698 {
1699     int i,j,k, rowTr;
1700     static unsigned int escStat = 0; // variable to hold current escape sequence status
1701     #define ESC_ON 0x00000001
1702     #define CSI_ON 0x00000002
1703     #define CSI_QUESTION_ON 0x00000004
1704     // TODO - add flags for multi-stage escapes
1705     #define CSI_NUM_MAX 25
1706     static unsigned int csi_nums[CSI_NUM_MAX];
1707     static unsigned int csi_nums_rec = 0;
1708     static unsigned int csi_digits = 0; // to hold digits until a separator is hit
1709
1710     keepCursorInBounds();
1711     for(i = 0; i < bytes; ++i)
1712     {
1713         if(escStat & CSI_ON)
1714         {
1715             // handle csi codes
1716             // if you get:
1717             // <CSI escape><number>m
1718             // you change colors. You can add multiple color options like this:
1719             // <CSI esc><number>;<number>m
1720             // Other CSI sequences behave similarly, a semicolon separated list of
decimal numbers

```

```

1724         // The function is determined by the terminating character (m for color
options, others for other things)
1725         // Note that the decimal number can be multiple bytes long... but probably
not more than 2.
1726         // My testing on the Linux console shows that numbers with more than 2
digits are to be simply ignored.

1727
1728         if (buf[i] >= 48 && buf[i] <= 57)
1729             // numbers in ascii are 48-57 (0-9)
1730             {
1731                 csi_digits *= 10; // decimal left shift
1732                 csi_digits += buf[i] - 48; // add in the new number
1733             }
1734         else if (buf[i] == ';')
1735             { // separator for another number
1736                 if (csi_nums_rec < CSI_NUM_MAX)
1737                 {
1738                     csi_nums[csi_nums_rec++] = csi_digits;
1739                 }
1740                 csi_digits = 0;
1741             }
1742         else if (buf[i] == '?')
1743             {
1744                 escStat |= CSI_QUESTION_ON;
1745             }
1746         else
1747             { // Here we handle the numbers received based on the terminating
character.
1748
1749                 // if no numbers have been received, treat it as if a 0 were
received
1750
1751                 csi_nums[csi_nums_rec++] = csi_digits;
1752                 csi_digits = 0;
1753
1754                 switch (buf[i])
1755                 {
1756                     case 'm': // Colors!!!
1757                         handle_colorCodes(csi_nums, csi_nums_rec);
1758                         break;
1759                     case 'A': // move cursor up n rows
1760                         for(j = 0; j < csi_nums_rec; ++j)
1761                         {
1762                             if(csi_nums[j] == 0)
1763                                 csi_nums[j] = 1;
1764                             moveCursor(csi_nums[j], UP);
1765                         }
1766                         break;
1767                     case 'B': // move down n rows
1768                     case 'e': // same as B
1769                         for(j = 0; j < csi_nums_rec; ++j)
1770                         {
1771                             if(csi_nums[j] == 0)
1772                                 csi_nums[j] = 1;
1773                             moveCursor(csi_nums[j], DOWN);
1774                         }
1775                         break;
1776                     case 'C': // move right n rows
1777                     case 'a': // same as C
1778                         for(j = 0; j < csi_nums_rec; ++j)
1779                         {
1780                             if(csi_nums[j] == 0)
1781                                 csi_nums[j] = 1;
1782                             moveCursor(csi_nums[j], RIGHT);
1783                         }
1784                         break;
1785                     case 'D': // move left n rows
1786                         for(j = 0; j < csi_nums_rec; ++j)
1787                         {
1788                             if(csi_nums[j] == 0)
1789                                 csi_nums[j] = 1;
1790                             moveCursor(csi_nums[j], LEFT);
1791                         }
1792                     }
1793                 }

```

```

1789     }
1790     break;
1791 case 'E': // move cursor down n rows, to column 1
1792     for(j = 0; j < csi_nums_rec; ++j)
1793     {
1794         if(csi_nums[j] == 0)
1795             csi_nums[j] = 1;
1796         moveCursor(csi_nums[j], DOWN);
1797     }
1798     cursorCol = 0;
1799     break;
1800 case 'F': // move cursor up n rows, to column 1
1801     for(j = 0; j < csi_nums_rec; ++j)
1802     {
1803         if(csi_nums[j] == 0)
1804             csi_nums[j] = 1;
1805         moveCursor(csi_nums[j], UP);
1806     }
1807     cursorCol = 0;
1808     break;
1809 case 'G': // move cursor to indicated column, current row
1810     remCursorAttr();
1811     cursorCol = csi_nums[csi_nums_rec-1] - 1;
1812     keepCursorInBounds();
1813     addCursorAttr();
1814     break;
1815 case 'H': // move cursor to indicated row,column
1816 case 'f': // same as H
1817     remCursorAttr();
1818     if (csi_nums_rec < 2)
1819         cursorCol = 0;
1820     else
1821         cursorCol = csi_nums[1] - 1;
1822     cursorRow = csi_nums[0] - 1;
1823
1824     keepCursorInBounds();
1825     addCursorAttr();
1826     break;
1827 case 'd': // move to indicated row, current column
1828     remCursorAttr();
1829     cursorRow = csi_nums[csi_nums_rec-1] - 1;
1830     if (cursorRow < 0)
1831         cursorRow = 0;
1832     else if (cursorRow > ROWS - 1)
1833         cursorRow = ROWS-1;
1834     addCursorAttr();
1835     break;
1836 case 'J':
1837     // Erase display
1838     if (csi_nums[csi_nums_rec-1] == 1)
1839     { // erase from start to cursor
1840         for(j = 0; j < cursorRow; ++j)
1841         {
1842             rowTr = (j + screenTop) % ROWS;
1843             for(k = 0; k < COLS; ++k)
1844             {
1845                 screenChars[rowTr][k] = ' ';
1846             }
1847         }
1848         for(j = 0; j < cursorCol; ++j)
1849         {
1850             screenChars[rowTr+1][j] = ' ';
1851         }
1852     }
1853     if (csi_nums[csi_nums_rec-1] == 0)
1854     { // erase from cursor to end
1855         for(j = cursorRow; j < ROWS; ++j)
1856         {
1857             for(k = 0; k < COLS; ++k)

```



```

1858                                     {
1859                                     rowTr = (j + screenTop) %
ROWS;
1860                                     screenChars[rowTr][k] = '
';
1861                                     }
1862                                     }
1863                                     rowTr = (cursorRow + screenTop) % ROWS;
1864                                     for(j = 0; j < cursorCol; ++j)
1865                                     {
1866                                     screenChars[cursorRow][j] = ' ';
1867                                     }
1868                                     }
1869                                     else if(csi_nums[csi_nums_rec-1] == 2 || csi_nums
[csi_nums_rec-1] == 3)
1870                                     { // erase whole display
1871                                     for(j = 0; j < ROWS; ++j)
1872                                     {
1873                                     for (k = 0; k < COLS; ++k)
1874                                     {
1875                                     screenChars[j][k] = ' ';
1876                                     }
1877                                     }
1878                                     break;
1879                                     case 'K':
1880                                     // Erase line
1881                                     rowTr = (cursorRow + screenTop) % ROWS;
1882                                     if (csi_nums[csi_nums_rec-1] == 1)
1883                                     { // erase from start to cursor
1884                                     for(j = 0; j < cursorCol; ++j)
1885                                     {
1886                                     screenChars[rowTr][j] = ' ';
1887                                     }
1888                                     }
1889                                     if (csi_nums[csi_nums_rec-1] == 0)
1890                                     { // erase from cursor to end
1891                                     for(j = cursorCol; j < COLS; ++j)
1892                                     {
1893                                     screenChars[rowTr][j] = ' ';
1894                                     }
1895                                     }
1896                                     else if(csi_nums[csi_nums_rec-1] == 2 || csi_nums
[csi_nums_rec-1] == 3)
1897                                     { // erase whole line
1898                                     for(j = 0; j < COLS; ++j)
1899                                     {
1900                                     screenChars[rowTr][j] = ' ';
1901                                     }
1902                                     }
1903                                     break;
1904                                     case 's': // push cursor position
1905                                     CSIpushCursor();
1906                                     break;
1907                                     case 'u': // pop cursor position
1908                                     CSIpopCursor();
1909                                     break;
1910                                     case 'n': // if we get 6, it reports cursor position
1911                                     if (csi_nums[0] == 6)
1912                                     {
1913                                     reportCursorToHost();
1914                                     }
1915                                     break;
1916                                     case 'l': // sometimes hides the cursor (ESC[?25l)
1917                                     if (csi_nums[0] == 25 && (escStat &
CSI_QUESTION_ON))
1918                                     {
1919                                     drawCursor = 0;
1920                                     }
1921                                     break;
1922

```

```

1923         case 'h': // sometimes shows the cursor (ESC[?25h)
1924             if (csi_nums[0] == 25 && (escStat &
CSI_QUESTION_ON))
1925             {
1926                 drawCursor = 1;
1927             }
1928             break;
1929         default:
1930             break;
1931     }
1932     escStat &= ~(CSI_ON | CSI_QUESTION_ON);
1933     csi_nums_rec = 0;
1934 }
1935 } //////////////// END HANDLING CSI CODES
1936 else if(escStat & ESC_ON)
1937 {
1938     // handle escape codes
1939 /*
1940     The following is an exerpt from the "console_codes" man page.
1941     We're not implementing all of them (Operating system command? Obviously
that's Linux only.)
1942
1943     ESC- but not CSI-sequences
1944
1945     ESC c      RIS      Reset.
1946     ESC D      IND      Linefeed.
1947     ESC E      NEL      Newline.
1948     ESC H      HTS      Set tab stop at current column.
1949     ESC M      RI       Reverse linefeed.
1950     ESC Z      DECID     DEC private identification. The kernel returns the
1951     string ESC [ ? 6 c, claiming that it is a VT102.
1952     ESC 7      DECSC     Save current state (cursor coordinates,
1953     attributes, character sets pointed at by G0, G1).
1954     ESC 8      DECRC     Restore state most recently saved by ESC 7.
1955     ESC [      CSI       Control sequence introducer
1956     ESC %       Start sequence selecting character set
1957     ESC % @     Select default (ISO 646 / ISO 8859-1)
1958     ESC % G     Select UTF-8
1959     ESC % 8     Select UTF-8 (obsolete)
1960     ESC # 8     DECALN   DEC screen alignment test - fill screen with E's.
1961     ESC (       Start sequence defining G0 character set
1962     ESC ( B     Select default (ISO 8859-1 mapping)
1963     ESC ( 0     Select VT100 graphics mapping
1964     ESC ( U     Select null mapping - straight to character ROM
1965     ESC ( K     Select user mapping - the map that is loaded by
1966     the utility mapscrn(8).
1967     ESC )       Start sequence defining G1
1968     (followed by one of B, 0, U, K, as above).
1969     ESC >       DECPNM   Set numeric keypad mode
1970     ESC =       DECPAM   Set application keypad mode
1971     ESC ]       OSC      (Should be: Operating system command) ESC ] P
1972     nrrggbb: set palette, with parameter given in 7
1973     hexadecimal digits after the final P :-(. Here n
1974     is the color (0-15), and rrggbb indicates the
1975     red/green/blue values (0-255). ESC ] R: reset
1976     palette
1977 */
1978 */
1979     switch(buf[i])
1980     {
1981         case '[':
1982             escStat |= CSI_ON;
1983             break;
1984         case 'c':
1985             // reset
1986             bufClear();
1987             break;
1988         case '7':
1989             ESCpushCursor();
1990             break;

```

```

1991         case '8':
1992             ESCpopCursor();
1993             break;
1994         case 'D':
1995         case 'E':
1996             handle_LF();
1997             break;
1998         case 'M':
1999             // reverse line feed...
2000             moveCursor(1,UP);
2001             break;
2002         case 'H':
2003             // set tab stop at current column...
2004             break;
2005         case 'Z':
2006             // report DECID
2007             break;
2008         default:
2009             break;
2010     }
2011     escStat &= ~ESC_ON;
2012 }
2013 else
2014 {
2015     switch(buf[i])
2016     {
2017         // Handle control characters
2018         //00 (NUL), 07 (BEL), 08 (BS), 09 (HT), 0a (LF), 0b (VT), 0c (FF),
2019         // 0d (CR), 0e (SO), 0f (SI), 18 (CAN), 1a (SUB), 1b (ESC), 7f
2020         (DEL)
2021         // 0x9B (CSI)
2022         case 0x00: // NUL
2023             break; // it's ignored.
2024         case 0x07: // BEL
2025             DAC_beep();
2026             break;
2027         case 0x08: // BS
2028             // This may be tricky... we may need so implement some
2029             sort of line discipline stuff to know how many characters the user's input since the last <return>
2030             // It works properly if user input doesn't make it go on
2031             to a second line... the same problem that plagues many
2032             // X terminal emulators...
2033             if (cursorCol > 0)
2034                 --cursorCol;
2035             keepCursorInBounds();
2036             break;
2037         case 0x09: // HT - goes to the next tab stop or to the end of the
2038             line if there is no earlier tab stop
2039             break;
2040         case 0x0A: // LF - down a line.
2041         case 0x0B: // VT, same as LF
2042         case 0x0C: // FF, same as VT and LF
2043             handle_LF();
2044             break;
2045         case 0x0D: // CR - push the carriage to the left! On old
2046             typewriters.
2047             handle_CR();
2048             break;
2049         case 0x0E: // SO - activate G1 character set
2050             break;
2051         case 0x0F: // SI - activate G0 character set
2052             break;
2053         case 0x18: // CAN - interrupt escape sequence (not sure what it
2054             does)
2055             break;
2056         case 0x1A: // SUB - interrupt escape sequence (not sure what it
2057             does)
2058             break;
2059         case 0x1B: // ESC - start escape sequence

```

```

2054         escStat |= ESC_ON;
2055         break;
2056     case 0x7F: // DEL - ignored
2057         break;
2058     case 0x9B: // CSI - start CSI sequence
2059         escStat |= CSI_ON;
2060         break;
2061     // Normal character handling!
2062     default:
2063         handle_normal(buf[i]);
2064         break;
2065     }
2066 }
2067 }
2068 }
2069
2070 // usart2.c
2071
2072 #include "usart2.h"
2073
2074 /*
2075     ; baud rate, 0x08, 12bit mantissa plus 4 bits fractional
2076     ; BRD = 8e6/(16*9600)= 52.083
2077     ; integer portion: int(52.083)=52 = 0b00110100      (0xD0 if 32Mhz, 0x138 for 48MHz)
2078     ; fractional portion: int(0.083*2^6+0.5)=6 = 0b0110   (0x32 if 32Mhz, 0x5 for 48MHz)
2079     ; to put in register: 0b001101000110 = 0x0341      (0xD03 if 32Mhz, 0x1385 for 48MHz)
2080     BAUD_RATE DCD 0x0341
2081
2082     ; control register 1, 0x0C, bits 31-14 reserved (0), 0b10000000001100 = 0x200C   no interrupts, no
2083     break, active receiver, RX and TX enabled
2084     USART_CTRL1_SETTINGS DCD 0x200C
2085
2086     SETUPA DCD 0x000A8AA8 ; Usart alt function settings -- see usart comments below
2087     ; alternate functionality pins
2088     ;USART2_CTS PA0 PD3 - input pull-up (8)
2089     ;USART2_RTS PA1 PD4 - alt push-pull (A)
2090     ;USART2_TX PA2 PD5 - alt push-pull
2091     ;USART2_RX PA3 PD6 - input pull up
2092     ;USART2_CK PA4 PD7 - alt push-pull
2093 */
2094
2095 unsigned char usart2_data[USART2_DATA_SIZE];
2096 int usart2_bytes_rec = 0;
2097
2098 void usart2_init(void)
2099 {
2100     RCC->APB1ENR |= RCC_APB1ENR_USART2EN;
2101     //GPIOA->CRL = (GPIOA->CRL & 0xFFFF0000) | 0x000A8AA8;
2102     GPIOA->CRL = (GPIOA->CRL & 0xFFFF0000) | 0x000B8BB8;
2103
2104     //USART2->BRR = 0x0341;
2105     USART2->BRR = 0x1385;
2106     USART2->CR1 = 0x202C; //Enable RXNEIE(bit 5) bit for receive interrupt
2107     NVIC->ISER[1] = NVIC_ISER_SETENA_6;
2108 }
2109
2110 void usart2_tx(unsigned char byte)
2111 {
2112     while(!(USART2->SR & USART_SR_TC));
2113
2114     USART2->DR = byte;
2115 }
2116
2117 void USART2_IRQHandler()
2118 {
2119     if (usart2_bytes_rec < USART2_DATA_SIZE)
2120     {

```

```

2123         usart2_data[usart2_bytes_rec] = USART2->DR;
2124         usart2_bytes_rec++;
2125     }
2126
2127     NVIC->ISER[1] = NVIC_ICPR_CLRPEND_6;
2128 }
2129
2130 int usart2_memcpy(unsigned char * dst)
2131 // Copies the usart2 buffer, then resets the usart2 buffer
2132 {
2133     int ret, i;
2134
2135     // turn off interrupts
2136     NVIC->ICER[1] = NVIC_ICER_CLRENA_6;
2137
2138     // copy buffer
2139     for(i = 0; i < usart2_bytes_rec; ++i)
2140     {
2141         dst[i] = usart2_data[i];
2142     }
2143     ret = usart2_bytes_rec;
2144     usart2_bytes_rec = 0;
2145
2146     // enable interrupts
2147     NVIC->ISER[1] = NVIC_ISER_SETENA_6;
2148
2149     return ret;
2150 }

```