Microprocessor Systems Final Project

A Wireless Game of Bulls and Cows

Presented to Mr. Jean-Samuel Chénard on December 3rd, 2009 by

Mathieu Perreault (260158758)
Logan Smyth (260179735)
Simon Foucher (260223197)
Alexandru Susma (260235940)

Contents

Executive Summary 3
- Overview 3
- Features 3
- Team Members 3
- Images 3

Hardware Overview 4

Software Architecture 6
- PS2 Keyboard and Hex Display via SPI slave 6
- Wireless Communication 6
- Display Control 7
  - 4-bit mode 7
  - Initialization Procedure 7
  - Sending Data 8
  - Helper functions 8
- User Interface 8
  - Initialization 8
  - Option 1: Hosting Player User Interface 9
  - Option 2: Joining Player User Interface 9
  - Common Stream: Game User Interface 9
- Game Logic, Reliable Wireless Communication and User Interface 9
  - Initialization 10
  - Host Stream 10
  - Join Stream 11
  - Common Game Stream and wireless protocol 11
- Flash Memory 13

Problems or Unresolved Issues 14
- Flash Problems 14
- McGumps Boards & CPLD Unit 14
- Interrupts 14
- Wireless 14

Conclusion 15

Bibliography 16

Appendix A – State Machine Diagram 17

Executive Summary

Overview

This document presents the design and implementation of a wireless platform capable of playing the cows and bulls game. The platform was built using the McGumps board, which incorporates a TI MSP430 microprocessor, an Altera MAX CPLD chip, 2 UART interfaces and RS-232 and Parallel ports. The platform also includes the eZ430-RF2500T low-power wireless module (commonly called the CC2500). The overall product is a self-contained unit that can wirelessly synchronize itself with another device, designed under the same wireless protocol, and enable the user to wirelessly play the cows and bulls game.

Features

- External LCD Character Display (2 lines of 24 characters)
- Integration of any PS/2-compatible keyboard via a second SPI slave
- Reliable wireless communications
- High-level finite-state machine software architecture
- Debug interface via an RS-232 connector and LED display on the board

Team Members

- Mathieu Perreault – Team Leader, Display integration
- Logan Smyth – Software Architect, PS/2 keyboard and wireless communication integration
- Simon Foucher –Game development, Reliable Wireless Protocol Implementation
- Alexandru Susma – Software Development, Memory and Game Development

Images

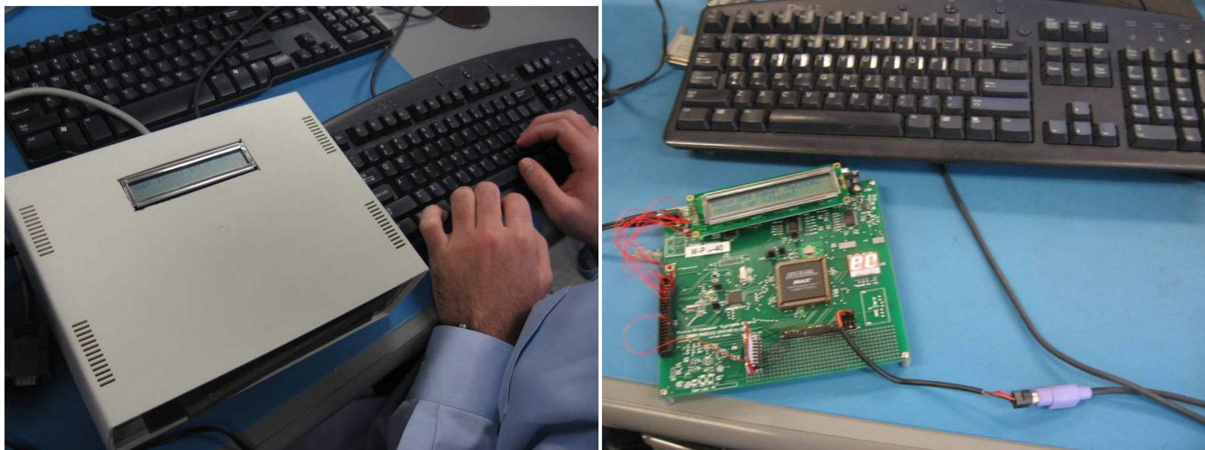


Figure 1 – (a) A user playing C&B using a PS/2 keyboard (b)Naked board showing peripherals and PS/2 connection

Hardware Overview

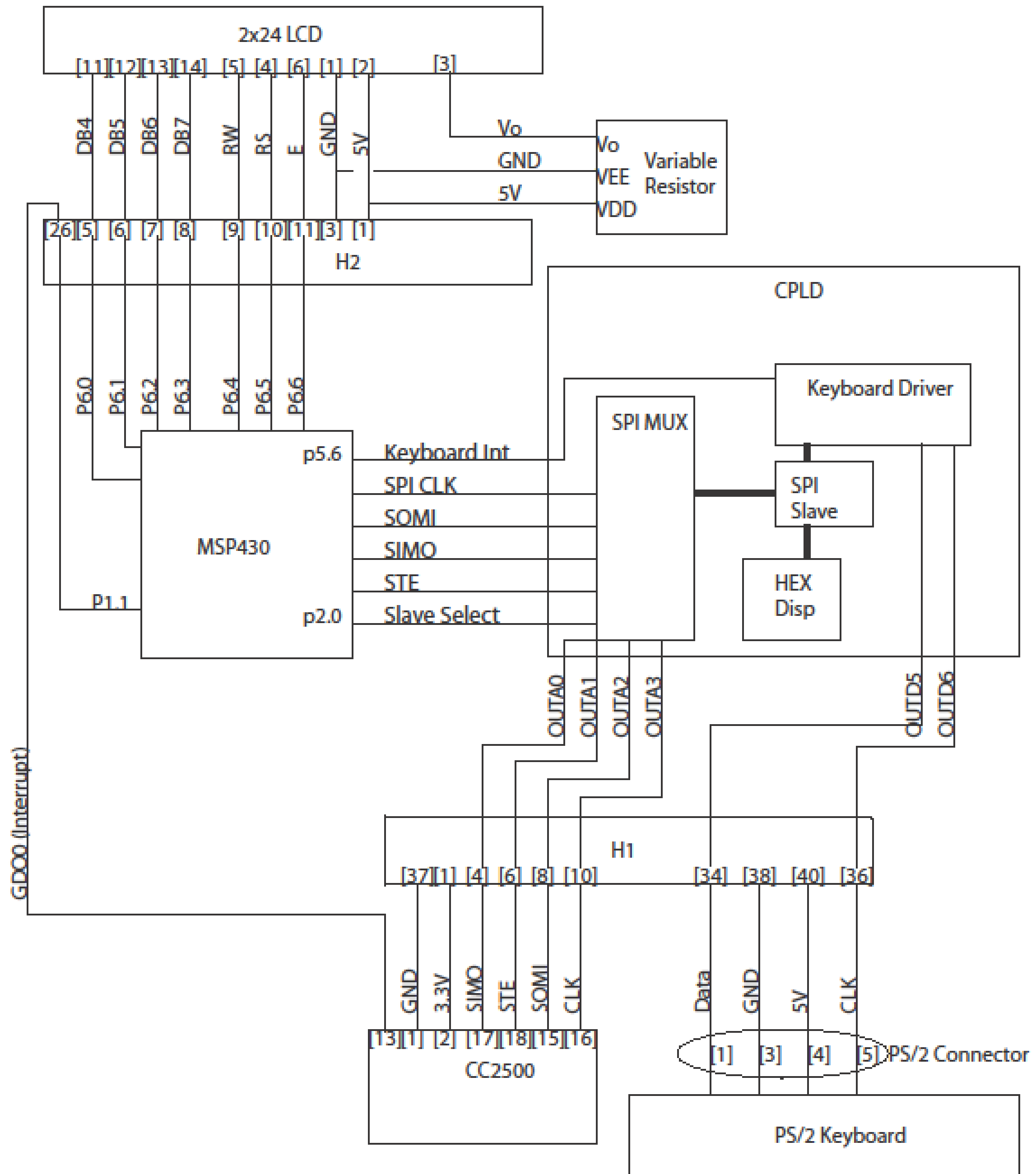


Figure 2 - Wiring diagram for the platform

Table 1 – Wirewrap Table for the whole platform

eZ430-RF2500T			McGumps	
PIN	NAME	to	PIN	NAME
17	SIMO		H1[04]	OUTA0
16	CLK		H1[10]	OUTA3
15	SOMI		H1[08]	OUTA2
18	STE		H1[06]	OUTA1
2	VCC		H1[01]	3.3V
1	GND		H1[37]	GND
13	GDO0		H2[26]	P1.1

2x24 LCD Character Display			McGumps	
PIN	NAME	to	PIN	NAME
11	DB4		H2[05]	P6.0
12	DB5		H2[06]	P6.1
13	DB6		H2[07]	P6.2
14	DB7		H2[08]	P6.3
5	RW		H2[09]	P6.4
4	RS		H2[10]	P6.5
6	E		H2[11]	P6.6
1	GND		H2[03]	GND
2	VDD		H2[01]	5V

Variable Resistor		to	McGumps	
PIN	NAME		PIN	NAME
	VDD		1	5V
	VEE		3	GND
Variable Resistor		to	2x24 LCD Char. Display	
	Vo		3	Vo

PS/2 Keyboard		to	McGumps	
PIN	NAME		PIN	NAME
1	Data		H1[34]	OUTD5
3	Ground		H1[38]	GND
4	+5V		H1[40]	5V
5	Clock		H1[36]	OUTD6

Software Architecture

PS2 Keyboard and Hex Display via SPI slave

Access to the PS2 keyboard and onboard seven-segment displays is accomplished using SPI. By making use of our preexisting SPI communication code and implementing an SPI slave in VHDL, we can very easily transfer data bidirectionally between the MSP430 and the CPLD.

We have offloaded much of the key press processing from the CPU onto the CPLD, allowing us to reduce the size of the interrupt routine. We began by implementing the SPI slave, which is a simple shift register, and then proceeded to develop the processing of the keyboard.

PS2 Keyboards return values of between 1 and 3 bytes depending on the key pressed, and each code relates to a specific key and action, such as press or release (Savard, 2007). We chose to use the VHDL to process these key codes, and then use an SPI transaction of 3 bytes to retrieve the values and convert them into ASCII inside the CPU. While this processing could have been done on the CPLD as well, the total size of the logic must be considered, since at the time we had also not implemented the hex display yet. When a key is pressed, the keyboard transmits the data serially with a start, stop and parity bit. When the last byte is received, the keyboard sends an interrupt to the CPU, will then read the scan codes and return an ASCII value.

Wireless Communication

The wireless is accessed via two methods, wireless_packet_transmit(packet* p) and wireless_packet_receive(packet* p). Transmit will write the contents of the packet into the TX FIFO one byte at a time and then set the cc2500 to TX mode. Receive simply reads a packet from a queue in memory and will return false if no packets are in the queue.

The system uses the receive interrupt to pull data in from the CC2500, which gives up several benefits.

Primarily it allows up to buffer more packets, which means we are less likely to drop values and wherefore increases reliability. Similarly, it reduces the time spent polling for packets because we simply check the queue instead of needing to do an SPI transaction. Another important benefit of this is that it allows us to abstract away several features such as heartbeat processing, which therefore reduces the complexity of the game loop. If we were to extend the protocol, this abstraction also would allow us to implement a reliable transmission protocol such as GoBackN transparently to the user.

We chose to store packet data in a single struct called packet, which contains a union for the body data. A union allows us to store several different data types in the same memory space, while at the same time keeping the values type-safe. This means that we do not need to cast back and forth between types and are protected by the standard compile time type checks. This also means that at transmission time, we simply have to cast our packet to a char* and iterate based on the length and read it back in the same fashion to restore an exact copy of the struct at the other end.

Display Control

This part will describe the implementation of a display controller using the MSP430. Specifically, the display controller was used to control the LUMEX “LCD-S401C40TF-1” 2x24 LCD Character Display used in this platform. The reference material was gathered from difference sources, the main of which is the datasheet of the LCD controller made by Samsung, the S6A0069 (Electronics, 2000). As shown in the Wiring Diagram in a previous section, the particular LCD connected to the MSP430 in this platform is connected to 9 pins of the H2 header of the McGumps board. Specifically, the four pins that are used to transmit data and the three pins that are used to control the type of data sent are connected to Port 6 of the MSP430. Furthermore, two more pins are connected to Ground (GND), and one to the +5V pin. The contrast of the LCD is controlled by a variable resistor. For more details about the connection, see Table 1, above. To implement some of the following procedures, some code was taken from the web (Nighsoft, 2009).

4-bit mode

It was decided that the implementation of the display controller in this platform would use the 4-bit mode. The main reason behind this choice was that Port 6 on the MSP 430 has only 8 pins whereas the 8-bit transaction mode of the Samsung controller would have required 8 data pins and 3 control pins, which exceeds the number available on Port 6. Instead of using part of another MSP430 port and because it required little more complexity, it was decided to use 4-bit mode.

Initialization Procedure

Functions: lcd_init(void)

Following the previously cited datasheet for the display controller, the initialization procedure, shown in Table 2, was performed on Port 6 of the MSP430. Our implementation would set the signals as appropriate for a transaction and pulse the “E” signal up and down, signaling the Samsung controller that valid data was on the line. Our implementation would also wait the required amount of time between different transactions, as specified by Figure 1. The different variables possible to configure the LCD were parameterized as follows:

```
DL = 0 : 4-bit mode
N = 1 : 2-line mode
F = 1 : Display ON
D = 1 : Display ON
C = 0 : Underline cursor OFF
B =  0: Cursor blink ON
SH = 0: Entire Shift OFF
ID = 1: Increment ON
```

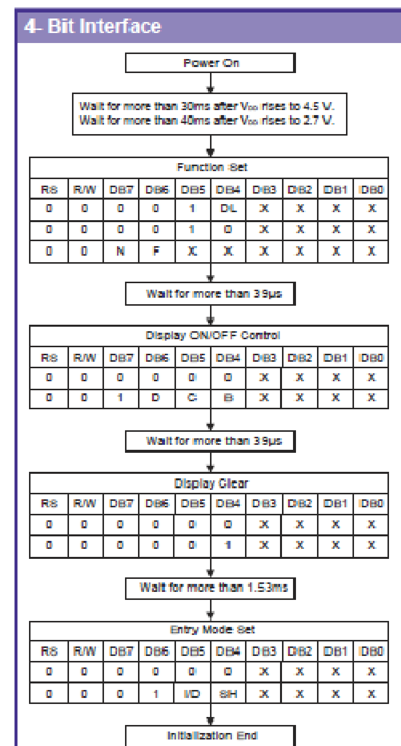


Figure 3 - 4-bit Initialization Sequence

Sending Data

Functions: send_data(char c)

To send 8-bit data to the display (in this case, printable characters), we set the 4 data lines to the most significant bits of our data first and pulse the "E" signal. Then, the 4 less significant bits are loaded on the appropriate lines are the "E" signal is probed again.

Helper functions

Building on the function to send data explained above, we have implemented a set of functions to control what is printed on the screen. The following is a brief explanation of the more important ones.

```
void lcd_putchar(char c);
```

This function takes as input a character and puts it on the screen at the current cursor location.

```
void lcd_goto(char p);
```

This function takes a position (0 to 47) and moves the cursor to that position on the LCD. It takes care of wrapping around to prevent writing off-screen.

```
void lcd_clearscreen(void);
```

This function clears the screen.

```
char lcd_printf(char* format, ...);
```

This function, adapted from work by Logan Smyth (Soft. Architect), emulates the standard printf function found in stdlib. It parses the different argument types normally found in printf (e.g. %d, %x, %s, %c, etc.) and prints on the LCD the proper rendering of the string.

```
void lcd_clearline(int line);
```

This function clears whatever line (1 or 2) it is passed as argument.

User Interface

Initialization

The main output device of our Cows and Bulls wireless platform is the 2x24 LCD Character Display. The following user interface description will assume output on this display device.

Upon boot-up of the system, users are greeted with a Welcome message and are asked to press Enter to get to start the game. They are then brought to the "Enter your name" screen, which they complete with the standard input device in this system, the PS/2 keyboard. Once the users enter their names and press the Enter key, they can press H or J, depending if they want to (H)ost or (J)oin a game, respectively. Any other key press will generate an error message. Based on whether the users are hosting or joining a game, the user interface will differ from Option 1 to Option 2, below:

Option 1: Hosting Player User Interface

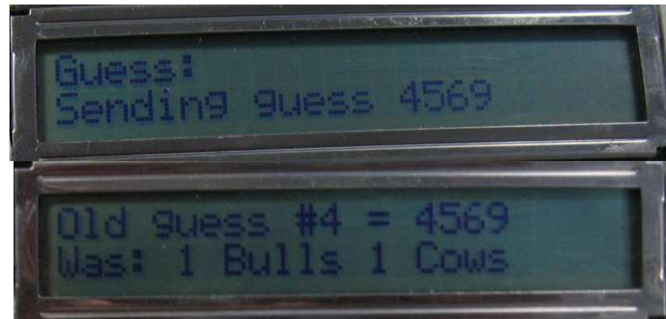
Once it is determined that the users want to host a game, the display informs the users that a connection is currently being set-up and sending reconnaissance beacons. When a join request comes in from a person trying to connect to the host, the users are shown a prompt on whether they want to accept (Y) or refuse (N) the connection from “user2,” where user2 is the name of the person wishing to join as they entered in the Initialization of the game. The User Interface then becomes common to both hosting and joining users as detailed in Game User Interface, below.

Option 2: Joining Player User Interface

Once it is determined that the users want to join a game, the display informs the users of an available game and sends a Join Request in the background to the host. Then, the User Interface will become common to both hosting and joining users as detailed in Game User Interface, below.

Common Stream: Game User Interface

Once users have entered the Game user interface, they will first be prompted for their own secret combination. As they type, the combination will be shown on the screen. They cannot enter the same digit twice in a combination (e.g “1123” is illegal). Once they press enter, they are brought the guess input screen. From there, they can type a guess for the other person’s secret number, and hit enter to send that guess. The Cows and Bulls count of that guess will be displayed a few seconds later when the result is received from the other party. Also, while in the guess input screen, users can press the TAB key repeatedly to go back and see the results of all the previous guesses they have made. Those results will be updated in the background as they are sent by the other player. To go back to the Guess Input screen, users should continue pressing TAB until the desired screen comes back up again (emulating a page shuffling or tab architecture). Once one of the users win the game by finding the correct combination, the correct message (You win, you lose) is displayed on each of the users’ device and shortly after, a Game Over message appears, prompting the users to press Enter to return to the host/join menu.



Game Logic, Reliable Wireless Communication and User Interface

The game was designed using a FSM, and both players use a very similar path in the state machine. The FSM starts at a common place then splits up into 2 streams while synchronizing players: one for hosting and one for joining. After the synchronization of both board, both player rejoin the ‘common’ game stream until the game is over. Once the game is over, the FSM will stall in its final state until a key is pressed, afterwards will move back to the initialization state to enable the player to restart a game. The game rules were taken from the web (Wikipedia, 2009).

The entire game flow is managed in the file ‘game.c’ with frequent function calls to driver files. Typically, when the FSM changes state, it sets a flag called ‘firstTime’ to TRUE. The next time the while loop

evaluates the game state, if this flag is set, the code will execute some state initialization script then set the flag to false. This could have been implemented by the use of function calls or intermediary states, but it was done in this manner in order to visually localize the code to facilitate understanding and debugging.

Initialization

Functions: playgame(void)
States: PRINT_INIT, HOST_OR_JOIN

PRINT_INIT

This is the initial state when the board is turned on and the game activated. The first time the state is visited, old guesses from previous games are removed (not deleted, but the pointers are set back to initial position), the heartbeats are turned off, the player name buffer is formatted and the player is prompted to press enter to begin then type his name.

When the function returns to this state, it will freeze in a getchar function until a key is presses. The ASCII key code will be recorded in a temporary buffer called 'keybuf' then interpreted. If the key pressed was a writable ASCII, it will be recorded in the player name string which will then get printed. If the player presses backspace, the last letter in player name string gets deleted, and if the player pressed 'enter' the game will display the recorded name then proceed.

We make the user of a keybuf buffer throughout the game such that we can filter between command and data keys. We can then interpret the content of the buffer and then either execute the command or record the data, based on the content of the buffer.

HOST_OR_JOIN

Simple state; prompts the user to hit 'h' to host or 'j' to join. If the user presses h/H, the game enters the host stream by changing to state 'SEND_HOST_BEACON'. If the user presses j/J, the game enters the join stream by going to state 'LISTEN_FOR_BEACON'. Any other key pressed will print an error message.

Host Stream

SEND_HOST_BEACON

Upon entering this state, a beacon packet is built in the packet struct 'beacon' then sent a first time, and the game time in seconds is recorded in the variable 'last_time'. Afterwards (and every time the game enters this state in the main while loop) the variable 'INTERVAL' records the timer ticks (range 0-32767). Then, if the game seconds have increased by 'GAME_HOST_BEACON_INTERVAL', the game records the new seconds count in 'last_time', waits 'INTERVAL' ticks then sends another beacon packet and prints a message on the screen with 'last_time' as a reference number such that the user can observe the packets being sent out. With this code, packets are sent at pseudo-random intervals: they will be sent out every 'GAME_HOST_BEACON_INTERVAL' seconds (fixed as a #define) + 'INTERVAL' timer ticks (which should change from one read to the other due to asynchronous interrupts). We can then manage the beacon interval while maintaining non-constant send times.

Once the beacon is sent, if any, all the packets in the packet queue will be read. If one of those packets is of type ‘R’, the name of the responder is printed on the screen and the game moves to the next state. Otherwise, the packet is discarded.

DO_YOU_ACCEPT

In this state, the host just received a join request from a player and is prompted if he accepts the game or not. The game freezes in a getchar function and key presses are recorded in a buffer. If the user presses y/Y, heartbeats are enabled (which will signal the joiner that his request got accepted), the other player’s address is recorded and the game enters the common game stream. If the user presses n/N, the game goes back to SEND_HOST_BEACON and keeps broadcasting. If the user presses ‘escape’, the user goes back to HOST_OR_JOIN.

Join Stream

LISTEN_FOR_BEACON

When entering this state, first we look for a timeout condition. We compare the variable last_time (which recorded the game seconds when the player pressed ‘j’ in HOST_OR_JOIN state) with the current game seconds and a timeout threshold. If the game time has passed the threshold, we assume that there are no games to be found; an error message is printed and the game goes back to HOST_OR_JOIN state.

Afterwards, we probe the packet queue to see if we received a beacon type packet. If so, we extract the other player’s name and display it on the screen. The source address of the received beacon is recorded and set as the packet target address and the state changes to ACCEPT_BEACON.

ACCEPT_BEACON

In this state, the board has found a game it could potentially join. The code sets up a beacon with this player’s name and sends it out. The game time in seconds is recorded and the state is changed to WAIT_HOST_HB.

WAIT_HOST_HB

Since a join request has been sent, if the other player accepts it he will enter that game mode and will start sending out heart beats. In this state, the joiner listens for a heart beat coming from the host he has contacted. There are 3 outcomes from this state. First, if a heart beet has not been found before a timeout, print an error message and go back to state HOSR_OR_JOIN. Second outcome, a heart beet has been received from the same address where the beacon was picked up, start the game by proceeding to state INIT_BOARD. In the third outcome, after waiting one second, if a heart beet has not yet been picked up but the join sequence has not timed out yet, the state is set to ACCEPT_BEACON, where another beacon will be sent out and then the game will come back here.

Common Game Stream and wireless protocol

Weather the player chooses to host or join a game, as soon as synchronization is completed, both players enter this stream the same way and play their game with the same code.

INIT_BOARD

This state is for the players to enter their secret numbers. First the buffers are reset, the heart Beats enabled (not done yet for the joiner) and the game pauses in a getchar function until the user presses a key which is stored in keybuf. Once a key is pressed, it is interpreted. If an invalid key is detected, a warning is printed and the key is ignored. The game goes back to the getchar function and waits for another key. If a valid digit from 0-9 is pressed, if it is not already present in the number the player is choosing, it is recorded and an up-to-date version of the player's number entry is printed on the screen. As per the game's requirements, the number is a character string, not an integer array. If the user presses backspace, the last entry of the player's number is erased from memory and his number is re-printed. Finally, if the user presses 'enter', if he has previously entered 4 digits in his number, the game proceeds to the GRAB_INPUT state, otherwise, the key press is ignored.

GRAB_INPUT

When this state is first entered, they key buffer is deleted since it still contains the player's number. Afterwards, if there are packets in the packet queue, they are interpreted by calling the function interpret_response. There are 2 types of packets this function is meant to interpret: guesses and responses to guesses since heartbeats and system packets are dealt with internally and do not appear in the packet queue. If we received a guess, we count the cows and bulls and send back an answer. If the other player found 4 bulls, the game prints 'you lose' on this player's screen and goes to state GAME_OVER. If we received a Y type packet, we record the cows and bulls in the previous guess buffer and print on the screen the result received. Here also, if this player guesses 4 bulls, 'you win' is printed and the game goes to GAME_OVER state.

Once all the packets in the packet queue have been interpreted, the code checks to see if we lost contact via heartbeats. If so, an error message is printed and the game goes back to PRINT_INIT state. Then the game stalls into a modified getchar function called getchartimeout. Since we are not running an OS to manage this multi-function environment, we need to free the CPU to check the packet queue every so often, so this function is like a getchat, except that it times out every 1/3 seconds, letting the program move forward. If during this period a key was pressed, it is recorded in keybuf. If the function times out, a timeout code is returned and recorded in keybuf. Afterwards, if keybuf has the timeout code, the function breaks, such that it goes back to checking the packet queue. Otherwise, we know that a key has been pressed and we interpret it.

If the user pressed enter, if he had previously entered a 4 character guess, the game moves to SEND_DATA. If the user presses backspace, the last character of the user's guess is deleted. If the user presses 'tab', his previous guesses get displayed sequentially (if any) on the screen. After all the guesses have been displayed, one more tab will bring back the guess screen. The old guesses are recorded in an array, and the game can record up to 30 guesses, after which, the buffer will start to overwrite the oldest guesses. Finally, if the user pressed a valid number, if it is not already present in his guess, it is recorded.

SEND_DATA

This state is entered when a guess is to be sent out. To manage wireless coherence, we use 2 variables: wait answer, which is a counter that decides how many guess packets are going to be sent out until we give up if no answer is received, and gotAnswer, which will be set to true by the interpret_response function of an answer to our guess got received. It would have been easier to simply have the guess number returned with the guess result, but this was not part of the protocol outlined guidelines, so here is how we worked around it.

This first thing done in this state is to interpret any packets in the buffer and interpret them. It might seem counter intuitive to look for a response before sending out a guess, but it is done this way such that the packet buffer gets emptied and any guess the other would have send is interpreted before we send out this guess. By doing so, we reduce the chances that the reply to this guess generates an overflow. Afterwards, if we haven't received an answer yet, we send out the guess and decrement the waitAnswer variable. Once this variable reaches zero, we assume that there is too much interference at the moment, print an error message and go back to GRAB_INPUT state. If we did received an answer to our guess, the flag gotAnswer got set to true in the interpret packet function, and the answer already got recorded in the oldGuessesResult buffer (but the pointer never got incremented, such that if we receive many answers to the same packet, they will be written to memory but without the pointer increment, will not affect the game). The guess send gets recorded in the oldGuesses buffer, and the oldGuesses pointer gets incremented such that the reception of any other responses to this guess will be written in an unused memory location. Once the response has been processes, the game goes back to the GRAB_INPUT state.

GAME_OVER

Simple state reached once the game is over. Buffers are reset, and when the user presses a key, the game goes back to HOST_OR_JOIN to enable a next game.

Flash Memory

For this project, the flash module of the MSP430 is used to store the values of all the configuration registers of the CC2500 Radio Frequency Chip. Once all the values are known, they are written to memory and read every time we reboot the system.

To implement flash memory access, both reading and writing, the information found in the User Manual proved to be very helpful. Understanding how the module works the flash timing generator was set and functions to perform writes and reads from memory were implemented. The starting address at which it was decided to store the registers is 0x1000 from the Flash Information Memory. Setting the registers of the flash controller we were able to perform erases, reads and writes. To write the values to memory a loop reading through all the CC2500 register was used. After every value was read it was written as a single byte to memory. For reading the same approach was taken.

Problems or Unresolved Issues

Flash Problems

Problems were encountered when writing the code for the flash memory access. Knowing that flash memory erase state is all 1s and realizing that before performing a write one needs to erase the memory, we included the erase code in a loop whose goal was to write multiple bytes to flash. The result was pretty strange since only the last byte was written. Checking again the way flash memory is divided it was realized that the smallest part one can erase at a time is a 512 bytes segment. Removing the erase from the loop made the function write all the bytes properly to flash. The module was tested on its own and worked in all cases, however because it was not until the last moment that we integrated the code in the complete system, we ran into some trouble. Putting the code together with the rest of the system seemed to create some interrupts to overlap while reading and writing through the SPI from the CC2500 chip. Due to the lack of time it never got fixed. Another design mistake made while implementing flash was the choice of address. Analyzing the memory segmentation we realized that flash memory actually starts at 0x1000. Reading the manual we found out that code and information can be located in both the Flash Main Memory and Flash Information Memory. Even though in our case it did not seem to interfere with the code, a good thing to do before choosing 0x1000 would have been to see if there was anything there to begin with.

McGumps Boards & CPLD Unit

During the entire semester, the CPLD chips were performing unreliably. Due to the socket design, the pins were sometimes not properly connected which made our code to act weird. To fix the problem we had to lightly poke the CPLD to make sure contacts were made. Early on, it was not clear what the problem was and the team spent a lot of time debugging the code, while the actual problem was within the board itself. Towards the end the poking did not work anymore so both boards had to eventually be replaced.

Interrupts

While working on the project we ran into a lot of problems caused by overlapping interrupts. Since our whole system is interrupt based we had different functions performed at different times triggered by interrupts. When interrupts would occur at the same time and basically interrupt each other, the program would freeze.

Wireless

Given the hardware we had available and the amount of interference present in the lab, a lot of time was spent on making the two boards communicate reliably. The biggest problem encountered while working on the basic send receive functions were the CRC Errors, which were caused most likely by the overlapping channels and interference from the McGill Wireless access points as well as from other teams working on the project.

Conclusion

The Wireless Cows and Bulls platform was an interesting project to implement. Our team appreciated the fact that the final project built upon modules developed in previous labs, therefore alleviating the tasks of developing all the drivers at once. The wireless protocol was simple enough to implement and the CC2500 was abstracting away a lot of complicated signal processing. The SPI interface was a good choice because of its versatility and ease of configuration. However, our team has found that the CPLD chip had limited resources. Some time was lost in trying to optimize the VHDL code such that it would be compiled and mapped in the available macrocells. Also, the poor quality of the CPLD sockets used made us waste a significant amount of time trying to debug the code, only to find out days later that the issue was actually with the socket connections. A word can be said about the Quartus software, which is numerous versions behind the latest one available (5.0 vs 9.1), especially considering that the latest version is installed in the Digital Systems Design lab in Trottier. Also, Rowley has released version 2.0 of its compiler, which bring better performance and more functionality. Overall, our team was satisfied with the results of this project. The actual competition format made us try to raise the bar as high as possible and I hope we made it to a certain extent.

Bibliography

Electronics, S. (2000, June). S6A0069 Datasheet.

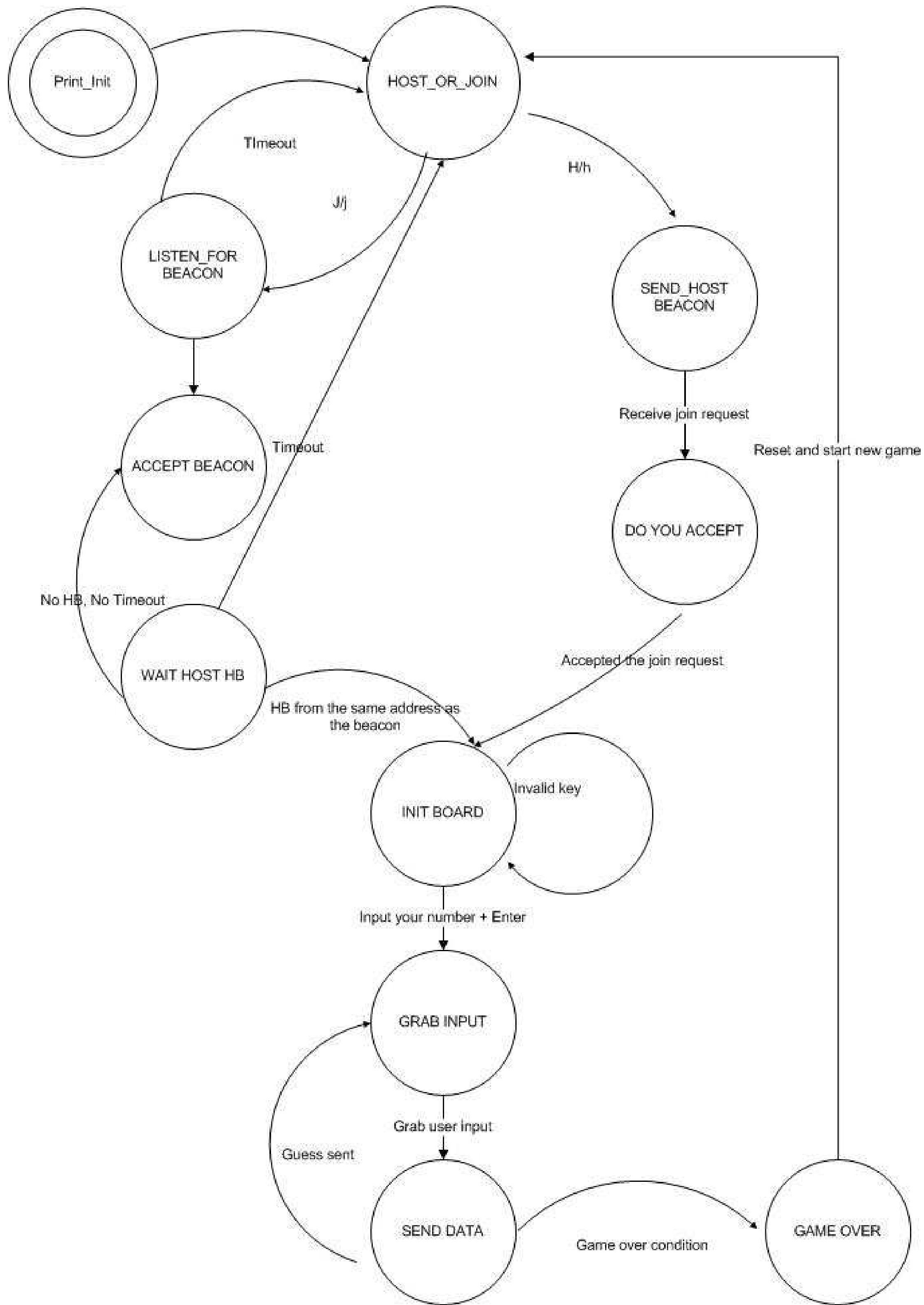
Nighsoft. (2009, Jan. 11). *Sparkfun Forums*. Retrieved 11 04, 2009, from Sparkfun: http://forum.sparkfun.com/viewtopic.php?t=13656

Savard, J. (2007). *Scan Codes Demystified*. Retrieved 12 2, 2009, from John Savard's Homepage: http://www.quadibloc.com/comp/scan.htm

Wikipedia. (2009). *Bulls and Cows*. Retrieved 12 2, 2009, from Wikipedia: http://en.wikipedia.org/wiki/Bulls_and_cows

And all the datasheets and specification documents we were provided with in the context of the class.

Appendix A – State Machine Diagram



```
  1:
  2: #include "cc2500.h"
  3: #include "cc2500_registers.h"
  4: #include "common.h"
  5:
  6: char cc2500_status_byte = 0x00;
  7:
  8: char cc2500_last_lqi_byte = 0;
  9:
 10:
 11: /**
 12:  * Initialize the CC2500
 13:  *
 14:  * Initialize UART, then reset and configure all registers
 15:  */
 16: void cc2500_init(void) {
 17:
 18:   uart_switch(CC2500_UART);
 19:   uart_init();
 20:
 21:   cc2500_reset();
 22:   timer_wait(10);
 23:   cc2500_read_reg(0x05);
 24:   cc2500_setradioregisters();
 25:
 26:
 27:   cc2500_strobe(CC2500REG_SFTX);
 28:   cc2500_strobe(CC2500REG_SFRX);
 29:   cc2500_strobe(CC2500REG_SRX);
 30: }
 31:
 32:
 33: /**
 34:  * Perform reset on CC2500 to put it in known state for future use
 35:  */
 36: void cc2500_reset(void){
 37:   P5SEL &= 0xF0; //Selecting control of the pins
 38:   P5DIR = 0xB; //1011 out
 39:
 40:   P5OUT = 0x09; //clock high and STE high
 41:
 42:   P5OUT &=~0x01; // STE low
 43:   timer_wait(10);
 44:
 45:   P5OUT |= 0x01; //STE high
 46:
 47:   timer_wait(10);
 48:   P5OUT &= ~0x01; // STE low
 49:
 50:   while(P5IN & 0x04) continue; //Wait until SOMI goes low
 51:
 52:   P5SEL |= 0x0E; //give control to UART for 1110
 53:   uart_spi_byte(0x30,SPI_SINGLE_TRANS); //Send reset strobe
 54:
 55:   //while(P5IN & 0x04) continue; //Wait until SOMI goes low
 56:   timer_wait(10);
 57:
 58: }
 59:
 60: /**
 61:  * Set up CC2500 using register configuration values provided by TAs
 62:  */
 63: void cc2500_setradioregisters(void){
```

```
  64:
  65:   cc2500_write_reg(CC2500REG_MCSM0 , 0x18);
  66:   cc2500_write_reg(CC2500REG_MCSM1 , 0x3f);
  67:
  68:   cc2500_write_reg(CC2500REG_FSCTRL1 , 0x07);
  69:   cc2500_write_reg(CC2500REG_FSCTRL0 , 0x00);
  70:   cc2500_write_reg(CC2500REG_FREQ2 , 0x5D);
  71:   cc2500_write_reg(CC2500REG_FREQ1 , 0x44);
  72:   cc2500_write_reg(CC2500REG_FREQ0 , 0xEC);
  73:   cc2500_write_reg(CC2500REG_MDMCFG4 , 0x2D);
  74:   cc2500_write_reg(CC2500REG_MDMCFG3 , 0x3B);
  75:   cc2500_write_reg(CC2500REG_MDMCFG2 , 0x73);
  76:   cc2500_write_reg(CC2500REG_MDMCFG1 , 0x23);
  77:   cc2500_write_reg(CC2500REG_MDMCFG0 , 0x3B);
  78:
  79:   // CHANNEL
  80:   cc2500_write_reg(CC2500REG_CHANNR , 0x03);
  81:   cc2500_write_reg(CC2500REG_DEVIATN , 0x01);
  82:   cc2500_write_reg(CC2500REG_FREND1 , 0xB6);
  83:   cc2500_write_reg(CC2500REG_FREND0 , 0x10);
  84:   cc2500_write_reg(CC2500REG_MCSM0 , 0x18);
  85:   cc2500_write_reg(CC2500REG_FOCCFG , 0x1D);
  86:   cc2500_write_reg(CC2500REG_BSCFG , 0x1C);
  87:   cc2500_write_reg(CC2500REG_AGCCTRL2 , 0xC7);
  88:   cc2500_write_reg(CC2500REG_AGCCTRL1 , 0x00);
  89:   cc2500_write_reg(CC2500REG_AGCCTRL0 , 0xB0);
  90:   cc2500_write_reg(CC2500REG_FSCAL3 , 0xEA);
  91:   cc2500_write_reg(CC2500REG_FSCAL2 , 0x0A);
  92:   cc2500_write_reg(CC2500REG_FSCAL1 , 0x00);
  93:   cc2500_write_reg(CC2500REG_FSCAL0 , 0x11);
  94:   cc2500_write_reg(CC2500REG_FSTEST , 0x59);
  95:   cc2500_write_reg(CC2500REG_TEST2 , 0x88);
  96:   cc2500_write_reg(CC2500REG_TEST1 , 0x31);
  97:   cc2500_write_reg(CC2500REG_TEST0 , 0x0B);
  98:   cc2500_write_reg(CC2500REG_FIFOTHR , 0x07);
  99:   cc2500_write_reg(CC2500REG_IOCFG2 , 0x29);
 100:   cc2500_write_reg(CC2500REG_IOCFG0 , 0x06); //Pin GD0 will go high (interrupt) and the
n low when it received the packet
 101:   cc2500_write_reg(CC2500REG_PKTCTRL1 , 0x04);
 102:   cc2500_write_reg(CC2500REG_PKTCTRL0 , 0x05);
 103:
 104:   // Address
 105:   cc2500_write_reg(CC2500REG_ADDR , 0xff);
 106:   cc2500_write_reg(CC2500REG_PKTLEN , 0xFF);
 107:
 108: //  cc2500_write_reg(CC2500REG_MCSM1 , 0x30);
 109:
 110: }
 111:
 112:
 113: /**
 114:  * Retreive status byte returned by last transaction
 115:  */
 116: char cc2500_status(void) {
 117:   return cc2500_status_byte;
 118: }
 119:
 120: int cc2500_verify(void) {
 121:   int i;
 122:   for(i = 0; i < 10; i++) {
 123:     if(cc2500_read_reg(CC2500REG_PARTNUM) != 0x80 || cc2500_read_reg(CC2500REG_VERSION)
 != 0x03) {
 124:       return FALSE;
```

```
125:     }
126:   }
127:   return TRUE;
128: }
129:
130: /**
131:  * Read a CC2500 register by sending an address with the read bit set, then sending a d
ummy byte and returning the values
132:  */
133: char cc2500_read_reg(char i) {
134:   uart_switch(CC2500_UART);
135:   cc2500_status_byte = uart_spi_byte(CC2500_READ | i, SPI_START_TRANS);
136:   return uart_spi_byte(0x00, SPI_END_TRANS);
137: }
138:
139:
140: /**
141:  * Write to a  CC2500 register byte sending an address with the write bit set and sendi
ng a second byte value
142:  */
143: void cc2500_write_reg(char i, char val) {
144:   uart_switch(CC2500_UART);
145:   cc2500_status_byte = uart_spi_byte(CC2500_WRITE | i, SPI_START_TRANS);
146:   uart_spi_byte(val, SPI_END_TRANS);
147: }
148:
149:
150: /**
151:  * Scan the register array to retreive the string describing the given address
152:  */
153: char* cc2500_reg_get_name(int addr) {
154:   int reg_size = sizeof(cc2500_reg);
155:   int i;
156:
157:   for(i = 0; i < sizeof(registers)/reg_size; i++) {
158:     if(registers[i].address == addr) {
159:       return registers[i].name;
160:     }
161:   }
162:   return (char*)0x0;
163: }
164:
165: /**
166:  * Scan the register array to retreive the address of the given address by name
167:  */
168: int cc2500_reg_get_address(char* name) {
169:   int reg_size = sizeof(cc2500_reg);
170:   int i;
171:
172:   for(i = 0; i < sizeof(registers)/reg_size; i++) {
173:     if(strcmp(registers[i].name, name) == 0) {
174:       return registers[i].address;
175:     }
176:   }
177:   return 0xFF;
178: }
179:
180: // Send command strobe
181: void cc2500_strobe(char strobe){
182:   uart_switch(CC2500_UART);
183:   uart_spi_byte(strobe, SPI_SINGLE_TRANS);
184: }
185:
```

```
186:
187: // Size of RX FIFO?
188: int cc2500_rx_size(void) {
189:   return cc2500_read_reg(CC2500REG_RXBYTES);
190: }
191:
192: // Size of TX FIFO?
193: int cc2500_tx_size(void) {
194:   return cc2500_read_reg(CC2500REG_TXBYTES);
195: }
196:
197: // Is in idle state?
198: int cc2500_state_idle(void){
199:   return (cc2500_state() == CC2500STATE_IDLE);
200: }
201:
202: // Is in receive state?
203: int cc2500_state_rx(void){
204:   int state = cc2500_state();
205:   return (state == CC2500STATE_RX0 ||state == CC2500STATE_RX1 ||state == CC2500STATE_RX
2);
206: }
207:
208: // Is in transmit state?
209: int cc2500_state_tx(void){
210:   int state = cc2500_state();
211:   return (state == CC2500STATE_TX0 ||state == CC2500STATE_TX1);
212: }
213:
214: // Get state
215: int cc2500_state(void) {
216:   return cc2500_read_reg(CC2500REG_MARCSTATE);
217: }
218:
219: // Flush RX Buffer
220: void cc2500_flush_rx(void) {
221:   if(cc2500_state() != CC2500STATE_RXOVERFLOW) {
222:     cc2500_strobe(CC2500REG_SIDLE);
223:   }
224:   cc2500_strobe(CC2500REG_SFRX);
225: }
226:
227: // Flush TX Buffer
228: void cc2500_flush_tx(void) {
229:   if(cc2500_state() != CC2500STATE_TXUNDERFLOW) {
230:     cc2500_strobe(CC2500REG_SIDLE);
231:   }
232:   cc2500_strobe(CC2500REG_SFTX);
233: }
234:
235:
236:
237: /**
238:  * Read from the RX FIFO into the given packet
239:  *
240:  * Returns FALSE on error
241:  * - If packet CRC Error
242:  * - If packet corrupted by FIFO read
243:  * - If overflown and some part missing
244:  */
245: int cc2500_read_packet(packet* p) {
246:   int i;
247:   char* buffer = (char*) p;
```

```
248:
249:   while(cc2500_rx_size() == 0) continue;
250:
251:   buffer[0] = cc2500_read_reg(CC2500REG_DATABUF);
252:
253:   i = cc2500_rx_size();
254:   // if rx < buffer and in overflow OR if this packet was corrupted
255:   // because of the problem on pg43 rbottom right
256:   if(buffer[0] > sizeof(packet)-1 || ((i&0x7F) < buffer[0] && (i&0x80) != 0)) {
257:     io_printf(IO_HYPERTERM, "Overflowed and missing data\r\n");
258:     return FALSE;
259:   }
260:   else if(cc2500_last_lqi_byte == buffer[0]){
261:     io_printf(IO_HYPERTERM, "Packet corrupted by bad read\r\n");
262:     return FALSE;
263:   }
264:
265:
266:   while(cc2500_rx_size() < buffer[0]+2) {
267:     continue;
268:   }
269:
270:   for(i = 0; i < buffer[0] && i < sizeof(packet)-3; i++) {
271:     buffer[i+1] = cc2500_read_reg(CC2500REG_DATABUF);
272:   }
273:   p->rssi = cc2500_read_reg(CC2500REG_DATABUF);
274:   p->lqi = cc2500_read_reg(CC2500REG_DATABUF);
275:   cc2500_last_lqi_byte = p->lqi;
276:   if(p->lqi&0x80){ //check CRC status bit
277:     return TRUE;
278:   }
279:   else {
280:     io_printf(IO_HYPERTERM, "CRC Error\r\n");
281:     return FALSE;
282:   }
283: }
284:
285:
286: /*
287:  * Write a given packet to the TX FIFO
288:  */
289: void cc2500_write_packet(packet* p) {
290:   int i;
291:   char* buffer = (char*) p;
292:
293:   for( i = 0; i <= buffer[0]; i++ ){  // "<=" to include length byte
294:     cc2500_write_reg(CC2500REG_DATABUF, buffer[i]);
295:   }
296: }
297:
298:
299: /**
300:  * Receives packet and stores it in the passed packet struct
301:  * Returns FALSE on error
302:  */
303: int cc2500_receive_packet(packet* p) {
304:   int stat;
305:
306:   // If not receiving currently
307:   if(!cc2500_state_rx()) {
308:     stat = cc2500_state();
309:     if(stat == CC2500STATE_IDLE){
310:       io_printf(IO_HYPERTERM, "Idle. Strobing rx\r\n");
```

```
311:       cc2500_strobe(CC2500REG_SRX);
312:       while(!cc2500_state_rx()) continue;
313:     }
314:     else if(stat == CC2500STATE_RXOVERFLOW){
315:       if(cc2500_rx_size() == 0) {
316:         //and already read all packets from buffer
317:         //Then go back to rx mode
318:         io_printf(IO_HYPERTERM, "Autoclearing and restarting RX\r\n");
319:         cc2500_flush_rx();
320:         cc2500_strobe(CC2500REG_SRX);
321:         while(!cc2500_state_rx()) continue;
322:       }
323:       else {
324:         io_printf(IO_HYPERTERM, "rx overflow with data in queue\r\n");
325: 
326:       }
327:     }
328:     else {
329:       //io_printf(IO_HYPERTERM, "Receive return false\r\n");
330:       return FALSE;
331:     }
332:   }
333: 
334:   stat = cc2500_read_packet(p);
335:   if(stat == FALSE) {
336:     //If the buffer didn't have enough bytes, assume an overflow and reset
337:     io_printf(IO_HYPERTERM, "Packet Overflowed buffer, flushing\r\n");
338:     cc2500_flush_rx();
339:     cc2500_strobe(CC2500REG_SRX);
340:     while(!cc2500_state_rx()) continue;
341:   }
342: 
343:   return stat;
344: }
345: 
346: 
347: /**
348:  * Transmits given packet
349:  *
350:  */
351: int cc2500_transmit_packet(packet* p) {
352:   int i;
353:   char dat;
354: 
355:   cc2500_write_packet(p);
356:   cc2500_strobe(CC2500REG_STX); //Put in transmit mode
357: 
358:   while(cc2500_state_tx()) continue;
359:   cc2500_strobe(CC2500REG_SRX); //having weird issues with it getting stuck in TX mode
360: 
361: //  io_printf(IO_HYPERTERM, "Done trans in state %i\r\n", cc2500_state());
362: 
363:   return TRUE;
364: }
365: 
366: 
367: /**
368:  * Returns true if their MIGHT be a packet available
369:  */
370: int cc2500_available_packet(void) {
371:   return (cc2500_rx_size() > 4); //going with 4 to cover length, addr, src and type
372: }
373: 
```

```
    1:
    2: #include <flash.h>
    3: #define FLASH_ADDRESS 1000
    4:
    5: void setFlashTimingGen(void)
    6: {
    7:   FCTL2 = FWKEY + FSSEL0 +FN4+FN2+FN1+FN0;            //divide MCLK
    8: }
    9: // Write one register value to flash memory
   10:
   11: void  writeToFlashMemory(char value, int offset)
   12: {
   13:   char *Flash_ptr;                            // Flash pointer
   14:
   15:   _DINT();
   16:
   17:   Flash_ptr = (char *)FLASH_ADDRESS+offset;
   18:   FCTL3 = FWKEY; // Clear Lock bit
   19:   FCTL1 = FWKEY + ERASE; // Set Erase bit
   20:  // *Flash_ptr = 0;                           //for debug use only - clear the flash
   21:
   22:   FCTL1 = FWKEY + WRT; // Set WRT bit for write operation
   23:
   24:    *Flash_ptr = value;
   25:
   26:   FCTL1 = FWKEY; // Clear WRT bit
   27:   FCTL3 = FWKEY + LOCK; // Set LOCK bit
   28:
   29:   _EINT();
   30: }
   31:
   32:
   33:
   34:
   35:
   36: // Read individual bytes from flash memory
   37:
   38: char readFromFlashMemory(int offset)
   39: {
   40:
   41: char *Flash_ptr,result;                              // Flash pointer
   42:
   43:   Flash_ptr = (char *)FLASH_ADDRESS+offset;
   44:
   45:   result = *Flash_ptr;
   46:   return result;
   47: }
```

```
    1: #include "game.h"
    2: #include "common.h"
    3: #include "io.h"
    4: #include "timer.h"
    5: #include "cc2500.h"
    6: #include "packet.h"
    7: #include "wireless.h"
    8: #include "hyperterm.h"
    9: #include "lcd.h"
   10:
   11: #define GAME_HOST_BEACON_INTERVAL 1
   12: #define GAME_CLIENT_BEACON_TIMEOUT 10
   13: #define GAME_HOST_ACCEPT_TIMEOUT  20
   14:
   15:
   16:
   17:   char number[4];
   18:
   19:   char oldGuesses[30][5];
   20:   int oldGuessesResults[30][2];// Records cows bulld
   21:   char key[5];
   22:   int oldGuessesPointer, scroll,waitAnswer, gotAnswer;
   23:
   24:
   25:
   26: /************************************************************
   27:   FSM implemented in playgame
   28:   Comments given on a state per state basis
   29: ************************************************************/
   30: void playGame(void){
   31:   packet beacon, HB, response, guess;
   32:
   33:
   34:   int STATE = PRINT_INIT, guessCounter = 0, i = 0, k,  firstTime = 1, foundBeacon = 0,
keyIndex, breakflag = 0;
   35:
   36:
   37:   char keybuf;
   38:
   39:   char THIS_PLAYER_NAME[20];
   40:   char OTHER_PLAYER_NAME[20];
   41:   int INTERVAL;
   42:
   43:   char guessMsg[4];
   44:
   45:
   46:
   47:   int game_host = FALSE;
   48:   int last_time = 0;
   49:
   50: //STATE = INIT_BOARD;
   51:
   52:   while(1){
   53:   switch(STATE){
   54:
   55:
   56: //STATE: PRINT_INIT-------------------------------------------
   57: //------ Game initialization: Print cow and prompt for user name
   58:     case PRINT_INIT:
   59:       if(firstTime){
   60:         oldGuessesPointer = 1;
   61:         scroll = 0;
   62:         wireless_heartbeat_disable();
```

```
   63:           // First format user name
   64:           for(i = 0; i < 20; i++)
   65:                 THIS_PLAYER_NAME[i] = '\0';
   66:
   67:           // Print logo
   68:           //io_printf(IO_HYPERTERM, "\n\n\n\r");
   69:           //io_printf(IO_HYPERTERM, "                    ______________________\n\r");
   70:           //io_printf(IO_HYPERTERM, "         (__)      /     COWS & BULLS     \\\n\r");
   71:           //io_printf(IO_HYPERTERM, "        ~(oo)~    (        Moooo!          )\n\r");
   72:           //io_printf(IO_HYPERTERM, "  /-------\\/  -----\\______________________/\n\r");
   73:           //io_printf(IO_HYPERTERM, " / |     ||\n\r");
   74:           //io_printf(IO_HYPERTERM, "*  ||----||\n\r");
   75:           //io_printf(IO_HYPERTERM, "   ^^    ^^\n\r");
   76:           //io_printf(IO_HYPERTERM, "Type your name (Enter accept, Escape Start over)\n\r
");
   77:           i = 0;
   78:
   79:
   80:           lcd_clearscreen();
   81:           lcd_printf("Welcome to Cows & Bulls!");
   82:           lcd_goto(LCD_LINE2);
   83:           lcd_printf("Press Enter to Start");
   84:           keybuf = io_getchar(IO_KEYBOARD);
   85:           if (keybuf != 0x0d) break;
   86:           lcd_clearscreen();
   87:           lcd_printf("Type your name: ");
   88:           lcd_goto(LCD_LINE2);
   89:           firstTime = 0;
   90:         }
   91:
   92:         // Grab user input for name, one character at a time
   93:         keybuf = io_getchar(IO_KEYBOARD);
   94:        // io_printf(IO_HYPERTERM, "Got %d\n\r", keybuf);
   95:
   96:         // Interpret user input
   97:         if(keybuf == 0x0d){         // User presses enter => move on
   98:           lcd_clearscreen();
   99:           lcd_printf("Recorded Name: %s", THIS_PLAYER_NAME);
  100:           STATE = HOST_OR_JOIN;
  101:           firstTime = 1;
  102:           break;
  103:         }
  104:
  105:         if(keybuf == 0x1b){         // User presses escape => reset
  106:           STATE = PRINT_INIT;
  107:           firstTime = 1;
  108:           break;
  109:         }
  110:
  111:
  112:         // Otherwise, record user name
  113:         if(i == 20){
  114:            //io_printf(IO_HYPERTERM, "Exceeded max name Length![enter to proceed, esc to
change\n\r");
  115:         }
  116:         else if (keybuf != 0x08){
  117:            lcd_putchar(keybuf);
  118:            THIS_PLAYER_NAME[i++] = keybuf;
  119:         }
  120:         else{
  121:           if (keyIndex != 0){
  122:             THIS_PLAYER_NAME[--i] = '\0';
  123:             lcd_clearline(2);
```

```
124:            lcd_printf(THIS_PLAYER_NAME);
125:          }
126:        }
127:
128:      break;
129:
130:
131: //STATE: HOST_OR_JOIN---------------------------------------------
132: //-------User prompt to be a game host or join a received beacon
133:      case HOST_OR_JOIN:
134:         if(firstTime){
135:              lcd_clearline(2);
136:              lcd_printf("(H)ost or (J)oin?");
137:              //io_printf(IO_HYPERTERM, "Do you want to host (h) or join (j)? ");
138:              firstTime = 0;
139:         }
140:
141:        // Grab user input and interpret it
142:        keybuf = io_getchar(IO_KEYBOARD);
143:        if (lcd_checkAlphabet(keybuf)) lcd_putchar(keybuf);
144:        switch(keybuf){
145:          case 'h':
146:          case 'H':
147:               STATE = SEND_HOST_BEACON;
148:               game_host = TRUE;
149:               firstTime = 1;
150:               packet_source(0x01);
151:               packet_destination(0x02);
152:          break;
153:
154:          case 'j':
155:          case 'J':
156:               STATE = LISTEN_FOR_BEACON;
157:               last_time = timer_get_seconds();
158:               game_host = FALSE;
159:               firstTime = 1;
160:               packet_source(0x02);
161:               packet_destination(0x01);
162:          break;
163:          default:
164:            lcd_printf(" (h/j)");
165:            timer_wait_seconds(1);
166:            lcd_clearline(2);
167:            lcd_printf("(H)ost or (J)oin?");
168:            //io_printf(IO_HYPERTERM, "Invalid selection!\n\r\n");
169:
170:        }
171:      break;
172:
173:
174:
175:
176:
177:
178: /*****************************************/
179: /*  Host Stream                          */
180: /*****************************************/
181:
182: //STATE:  SEND_HOST_BEACON---------------------------------------------
183: //-------Selects a random time and send beacons at random intervals
184:      case SEND_HOST_BEACON:
185:
186:        // Select random interval
```

```
187:        if(firstTime){
188:          timer_wait_seconds(1);
189:          lcd_clearline(2);
190:          lcd_printf("Setting up beacon...");
191:          //io_printf(IO_HYPERTERM, "\n\rSetting up beacon...\n\r");
192:          INTERVAL = timer_get_ticks();
193:          packet_create_h(&beacon, THIS_PLAYER_NAME);
194:          last_time = timer_get_seconds();
195:          cc2500_transmit_packet(&beacon);
196:          firstTime = 0;
197:        }
198:        if(timer_get_seconds() >= last_time + GAME_HOST_BEACON_INTERVAL ) {
199:          //timer_wait(timer_get_ticks());
200:          last_time = timer_get_seconds();
201:          cc2500_transmit_packet(&beacon);
202:          lcd_clearline(2);
203:          lcd_printf("Sending beacon ref#%d", last_time);
204:
205:        }
206:
207:        while(wireless_packet_available()) {
208:          // Look for response => Function returns 1 if response is received
209:          if(wireless_packet_receive(&response) && response.type == PACKET_R){
210:            lcd_clearscreen();
211:            lcd_printf("Answer fr: %s", response.body.r.name);
212:            //io_printf(IO_HYPERTERM, "Received answer from: %s", response.body.r.name);
213:            strncpy(OTHER_PLAYER_NAME, response.body.r.name, 20);
214:            lcd_goto(LCD_LINE2);
215:            lcd_printf("Accept game (y/n): ");
216:            //io_printf(IO_HYPERTERM, "\n\rAccept game (y/n): ");
217:            STATE = DO_YOU_ACCEPT;
218:            break;
219:          }
220:        }
221:      break;
222:
223:
224:
225: //STATE:  DO_YOU_ACCEPT---------------------------------------------
226: //-------Prompt if user accepts client
227:      case DO_YOU_ACCEPT:
228:        // Grab user input and interpret it (y/n)
229:        keybuf = io_getchar(IO_KEYBOARD);
230:        if (lcd_checkAlphabet(keybuf)) lcd_putchar(keybuf);
231:
232:        switch(keybuf){
233:          case 0x1b://escape
234:                STATE = HOST_OR_JOIN;
235:                lcd_clearscreen();
236:                lcd_printf("cancelling...");
237:                timer_wait_seconds(1);
238:                lcd_clearscreen();
239:                //io_printf(IO_HYPERTERM, "\n\r");
240:                firstTime = 1;
241:            break;
242:          case 'y':
243:          case 'Y':
244:               lcd_clearscreen();
245:               lcd_printf("You accepted the game");
246:               timer_wait_seconds(1);
247:               //io_printf(IO_HYPERTERM, "\n\rYou accepted the game\n\r\n");
248:               wireless_heartbeat_enable();
249:               STATE = INIT_BOARD;
```

```
  250:                firstTime = 1;
  251:           break;
  252: 
  253:           case 'n':
  254:           case 'N':
  255:                lcd_clearscreen();
  256:                lcd_printf("Refusing host; going back to broadcasting");
  257:                timer_wait_seconds(1);
  258:                lcd_clearscreen();
  259:                //io_printf(IO_HYPERTERM, "\n\rRefusing host; going back to broadcasting (
esc to quit)\n\r\n");
  260:                STATE = SEND_HOST_BEACON;
  261:                firstTime = 1;
  262:           break;
  263:           default:
  264:             lcd_printf(" Invalid (y/n)");
  265:             timer_wait_seconds(1);
  266:             lcd_clearline(2);
  267:             lcd_printf("Accept game (y/n): ");
  268:             //io_printf(IO_HYPERTERM, "\n\rInvalid selection!\n\r\n");
  269:         }
  270: 
  271:       break;
  272: 
  273: 
  274: 
  275: /******************************************/
  276: /*  Join Stream                           */
  277: /******************************************/
  278: 
  279: //STATE:  LISTEN_FOR_BEACON---------------------------------------------
  280: //-------Listens for a beacon and displays it on the screen
  281:      case LISTEN_FOR_BEACON:
  282:        if(timer_get_seconds() >= last_time + GAME_CLIENT_BEACON_TIMEOUT ) {
  283:          lcd_clearline(2);
  284:          lcd_printf("Found no games to join!");
  285:          //io_printf(IO_HYPERTERM, "Found no games to join \r\n");
  286:          timer_wait_seconds(1);
  287:          STATE = HOST_OR_JOIN;
  288:          firstTime = 1;
  289:        }
  290:        else if(wireless_packet_available() && wireless_packet_receive(&response) && resp
onse.type == PACKET_H){
  291:          lcd_clearscreen();
  292:          lcd_printf("Received Beacon:");
  293:          lcd_goto(24);
  294:          lcd_printf("%s", response.body.h.name);
  295:          //io_printf(IO_HYPERTERM, "Received the following Beacon: \n\r");
  296:          packet_print(IO_HYPERTERM, &response);
  297:          STATE = ACCEPT_BEACON;
  298:          last_time = timer_get_seconds();
  299:        }
  300:      break;
  301: 
  302: //STATE:  ACCEPT_BEACON---------------------------------------------
  303: //-------User accepts a beacon - send ACK packet
  304:      case ACCEPT_BEACON:
  305:          packet_create_r(&beacon, THIS_PLAYER_NAME);
  306:          cc2500_transmit_packet(&beacon);
  307: 
  308:          packet_print(IO_HYPERTERM, &beacon);
  309:          STATE = WAIT_HOST_HB;
  310:          last_time = timer_get_seconds();
```

```
 311:     break;
 312:
 313: //STATE:  WAIT_HOST_HB--------------------------------------------
 314: //-------Wait to receive host heartbeats and go to start game
 315:     case WAIT_HOST_HB:
 316:
 317:       if(timer_get_seconds() >= last_time + GAME_HOST_ACCEPT_TIMEOUT ) {
 318:           lcd_clearscreen();
 319:           lcd_printf("Host failed to accept your request");
 320:           //io_printf(IO_HYPERTERM, "Host failed to accept your request \r\n");
 321:           timer_wait(16000);
 322:           STATE = HOST_OR_JOIN;
 323:
 324:       }
 325:       else if(timer_get_seconds() < wireless_heartbeat_time() + 1){
 326:         STATE = INIT_BOARD;
 327:         firstTime = 1;
 328:       }
 329:       else{
 330:         timer_wait(30000);
 331:         STATE = ACCEPT_BEACON;
 332:       }
 333:
 334:     break;
 335:
 336:
 337:
 338:
 339: /*****************************************/
 340: /*  Play game (common) Stream            */
 341: /*****************************************/
 342:     // Reset buffers and prompt user to enter number
 343:     case INIT_BOARD:
 344:       if(firstTime){
 345:
 346:         wireless_heartbeat_enable();
 347:
 348:         keyIndex = 0;
 349:         lcd_clearscreen();
 350:         lcd_printf("Type 4 digit secret #:");
 351:         lcd_goto(24);
 352:         //io_printf(IO_HYPERTERM, "Type your 4 digit secret number [Enter when done]\n\
r");
 353:         firstTime = 0;
 354:         for(i = 0; i < 4; i++){ //Initiate the key array to \0 character
 355:           key[i] = '\0';
 356:         }
 357:       }
 358:
 359:      // If a key is pressed, record it and update LCD to reflect changes
 360:         keybuf = io_getchar(IO_KEYBOARD);
 361:
 362:       if(keybuf == 0x0d){ //user presses Enter the last 4 digit are kept and go to GRAB
_INPUT
 363:           if (keyIndex == 4){
 364:           STATE = GRAB_INPUT;
 365:           lcd_clearscreen();
 366:           lcd_printf("Your secret # is: %c%c%c%c", number[0], number[1], number[2], num
ber[3]);
 367:           //io_printf(IO_HYPERTERM, "Your secret number is: %c%c%c%c\n\r", number[0], n
umber[1], number[2], number[3]);
 368:           timer_wait_seconds(1);
 369:           firstTime = 1;
```

```
  370:            }
  371:        }
  372:        else if (keybuf == 0x08 && keyIndex != 0){
  373:           key[--keyIndex] = '\0'; //remove character from buffer
  374:
  375:            /*Print the buffer*/
  376:            //io_printf(IO_HYPERTERM, "\r");
  377:            lcd_clearline(2);
  378:            for(i = 0; i < 4; i++){
  379:              number[i] = key[i];
  380:              lcd_printf("%c", key[i]);
  381:              //io_printf(IO_HYPERTERM, "%c", key[i]);
  382:            }
  383:            //io_printf(IO_HYPERTERM, "\r");
  384:        }
  385:        else if(keybuf > '9' || keybuf < '0'){
  386:          lcd_clearscreen();
  387:          lcd_printf("Invalid key entered [%c]!Must be in range [0-9]", keybuf);
  388:          timer_wait_seconds(1);
  389:          lcd_clearscreen();
  390:          lcd_printf("Type 4 digit secret #:");
  391:          lcd_goto(LCD_LINE2);
  392:          for (i = 0; i < 4; i++) if (key[i] != '\0') lcd_putchar(key[i]);
  393:          //io_printf(IO_HYPERTERM, "Invalid key entered [%c]! Must be in the range [0-9]
\n\r", keybuf);
  394:        }
  395:        else { //the key is valid, check if it's already in the buffer and else put it in
 and print the buffer
  396:          for (i = 0; i < 4; i++){ //if the number is already in the combination, do noth
ing
  397:            if (keybuf == key[i] || keyIndex == 4) {
  398:              breakflag = 1; //key already present in the buffer !
  399:              break;
  400:            }
  401:            else breakflag = 0;
  402:          }
  403:          if (breakflag == 1) {
  404:            breakflag = 0;
  405:            break;
  406:            }
  407:          /*Put the key in the buffer*/
  408:          key[keyIndex++] = keybuf;
  409:
  410:          // Print the number on the LCD
  411:          //io_printf(IO_HYPERTERM, "\r");
  412:          lcd_clearline(2);
  413:          for(i = 0; i < 4; i++){
  414:            number[i] = key[i];
  415:            lcd_printf("%c", key[i]);
  416:            //io_printf(IO_HYPERTERM, "%c", key[i]);
  417:          }
  418:          //io_printf(IO_HYPERTERM, "\r");
  419:        }
  420:
  421:        break;
  422:
  423:        // Grab user input a when presses 'Enter', send data
  424:      case GRAB_INPUT:
  425:        if(firstTime){
  426:          lcd_clearscreen();
  427:          lcd_printf("Guess:");
  428:          lcd_goto(LCD_LINE2);
  429:          //io_printf(IO_HYPERTERM, "\n\rType guess for other player's number[Enter to se
```

```
nd]\n\r");
  430:          firstTime = 0;
  431:          keyIndex = 0;
  432:          for(i = 0; i < 4; i++){ //Initiate the key array to \0 character
  433:            key[i] = '\0';
  434:          }
  435:        }
  436:
  437:        while (wireless_packet_available()) {
  438:          wireless_packet_receive(&response);
  439:          interpret_response(&response, &STATE);
  440:          if(STATE == GAME_OVER)
  441:            break;
  442:          //firstTime = 1;
  443:        }
  444:
  445:        if (!wireless_heartbeat_status()){
  446:          lcd_clearscreen();
  447:          lcd_printf("Lost Connection");
  448:          timer_wait_seconds(2);
  449:          firstTime = 1;
  450:          STATE = PRINT_INIT;
  451:        }
  452:        //hyperterm_set_echo(FALSE);
  453:        keybuf = io_timeoutgetchar(IO_KEYBOARD);
  454:        //hyperterm_set_echo(TRUE);
  455:        if (keybuf == 0x11) break;   //if the user didn't press anything loop over (0x11
is our escape sequence to detect no change)
  456:
  457:        // User presses 'enter'; send the data and record guess
  458:        if(keybuf == 0x0d){
  459:          if (keyIndex == 4){
  460:            STATE = SEND_DATA;
  461:            firstTime = 1;
  462:          }
  463:        }
  464:        else if (keybuf == 0x08 && keyIndex != 0){
  465:           key[--keyIndex] = '\0'; //remove character from buffer
  466:
  467:            /*Print the buffer*/
  468:            //io_printf(IO_HYPERTERM, "\r");
  469:            lcd_clearline(2);
  470:            for(i = 0; i < 4; i++){
  471:              number[i] = key[i];
  472:              lcd_printf("%c", key[i]);
  473:             // io_printf(IO_HYPERTERM, "%c", key[i]);
  474:            }
  475:            //io_printf(IO_HYPERTERM, "\r");
  476:        }
  477:
  478:        else if(keybuf == 0x09){ //TAB key to scroll
  479:          if(oldGuessesPointer == 1){ //we have no guesses so disregard TAB get another c
haracter
  480:            lcd_clearscreen();
  481:            lcd_printf("You haven't made a guess yet!");
  482:            timer_wait_seconds(1);
  483:            lcd_clearscreen();
  484:            lcd_printf("Guess:");
  485:            lcd_goto(LCD_LINE2);
  486:            for (i = 0; i < 4; i++) if (key[i] != '\0') lcd_putchar(key[i]);
  487:            break;
  488:          }
  489:
```

```
 490:          // Otherwise, if we have some old guesses, print them out on the screen
 491:          if(scroll == -1){
 492:            scroll = oldGuessesPointer - 1;
 493:            break;
 494:          }
 495:          lcd_clearscreen();
 496:          lcd_printf("Old guess #%d = %c%c%c%c", scroll, oldGuesses[scroll][0],oldGuesses
[scroll][1],oldGuesses[scroll][2],oldGuesses[scroll][3]);
 497:          lcd_goto(24);
 498:          lcd_printf("Was: %d Bulls %d Cows", oldGuessesResults[scroll][0], oldGuessesRes
ults[scroll][1]);
 499:          if(--scroll < 0){
 500:            scroll = oldGuessesPointer-1;
 501:            lcd_clearscreen();
 502:            lcd_printf("Guess:");
 503:            lcd_goto(LCD_LINE2);
 504:            for (i = 0; i < 4; i++) if (key[i] != '\0') lcd_putchar(key[i]);
 505:            break;
 506:          }
 507:          break;
 508:        }
 509:
 510:        else if(keybuf > '9' || keybuf < '0'){
 511:          lcd_clearscreen();
 512:          lcd_printf("Invalid key entered [%c]!Must be in range [0-9]", keybuf);
 513:          timer_wait_seconds(1);
 514:          lcd_clearscreen();
 515:          lcd_printf("Guess:");
 516:          lcd_goto(LCD_LINE2);
 517:          for (i = 0; i < 4; i++) if (key[i] != '\0') lcd_putchar(key[i]);
 518:          //io_printf(IO_HYPERTERM, "Invalid key entered [%c]! Must be in the range [0-9]
\n\r", keybuf);
 519:          //io_printf(IO_HYPERTERM, "\n\rType guess for other player's number[Enter to se
nd]\n\r");
 520:        }
 521:        else if (keybuf == 0x1b){
 522:          lcd_clearscreen();
 523:          lcd_printf("Guess:");
 524:          lcd_goto(LCD_LINE2);
 525:          for (i = 0; i < 4; i++) if (key[i] != '\0') lcd_putchar(key[i]);
 526:        }
 527:        else  {
 528:          for (i = 0; i < 4; i++){ //if the number is already in the combination, do noth
ing
 529:            if (keybuf == key[i] || keyIndex == 4) {
 530:              breakflag = 1; //key already present in the buffer !
 531:              break;
 532:            }
 533:            else breakflag = 0;
 534:          }
 535:          if (breakflag == 1) {
 536:            breakflag = 0;
 537:            break;
 538:            }
 539:          /*Put the key in the buffer*/
 540:          key[keyIndex++] = keybuf;
 541:
 542:          lcd_clearscreen();
 543:          lcd_printf("Guess:");
 544:          lcd_goto(LCD_LINE2);
 545:          for (i = 0; i < 4; i++) if (key[i] != '\0') lcd_putchar(key[i]);
 546:        }
 547:      break;
```

```
548:
549:
550:
551:     case SEND_DATA:
552:        //Here we have the guess in position key[0],1,2,3 in char values from the keyboa
rd
553:       /*Create the packet with the contents of key*/
554:       if(firstTime){
555:         waitAnswer = 6;// Flag to notify that we are still waiting for an answer from o
ther player
556:         gotAnswer = FALSE;
557:         firstTime = FALSE;
558:       }
559:       while (wireless_packet_available()) {
560:         wireless_packet_receive(&response);
561:         interpret_response(&response, &STATE);
562:       }
563:
564:       if(!gotAnswer){
565:         packet_create_g(&guess, guessCounter, key);
566:         guessCounter++;
567:         cc2500_transmit_packet(&guess); //Off you go !
568:
569:         // Print what was sent and go waiting for more user input
570:         lcd_clearline(2);
571:
572:         lcd_printf("Sending guess %c%c%c%c", key[0], key[1], key[2], key[3]);
573:
574:         //io_printf(IO_HYPERTERM, "Sending: %c%c%c%c\n\r", key[0], key[1], key[2], key[
3]);
575:         timer_wait_seconds(1);
576:
577:         // Sent 6 packets but never got an answer
578:         if(--waitAnswer == 0){
579:           lcd_clearscreen();
580:           lcd_printf("Never got an answer for guess!");
581:           timer_wait_seconds(1);
582:           STATE = GRAB_INPUT;
583:           firstTime = 1;
584:         }
585:       }
586:       // If we did get an answer
587:       else{
588:         if(STATE == GAME_OVER)
589:           break;
590:         // Record guess in old guesses buffer only if we got an answer
591:         // Interpret function already recorded the cows and bulls.
592:         // Still need to record our guess, and increment pointer
593:         // OldGuessPointer points to the first empty spot in the array
594:         for(i = 0; i < 4; i++){
595:           oldGuesses[oldGuessesPointer][i] = key[i];
596:         }
597:
598:         scroll = oldGuessesPointer;
599:         if(++oldGuessesPointer == 30)
600:           oldGuessesPointer = 0;
601:         STATE = GRAB_INPUT;
602:         firstTime = 1;
603:         }
604:     break;
605:
606:     case GAME_OVER: //Print game over and go to init_board state !
607:       lcd_clearscreen();
```

```
  608:       lcd_printf("GAME OVER!!!");
  609:       lcd_goto(24);
  610:       lcd_printf("Press any key to reset");
  611:       io_getchar(IO_KEYBOARD);
  612:       STATE = HOST_OR_JOIN;
  613:       firstTime = 1;
  614:       lcd_clearscreen();
  615:       wireless_heartbeat_disable();
  616:       break;
  617:       //io_printf(IO_HYPERTERM, "Game is over fellas!\n\r");
  618:     default:
  619:       STATE = INIT_BOARD;
  620:     }
  621:   }
  622: }
  623: 
  624: 
  625: /************************************************************/
  626: /*  interpret_response(packet *response)
  627: /*      Function called in the game while loop while waiting for user input
  628: /*       If so: depending on packet type it sends either y packet or print bulls and co
ws
  629: /*      computes: Matching = Bulls, Close = cows
  630: /*      Fires answer to other player
  631: /************************************************************/
  632: void interpret_response(packet *response, int *current_state){
  633:  int i, k;
  634:  packet y_ack;
  635:  char received_guess[4];
  636:  int cows = 0, bulls = 0;
  637: 
  638:  //depending on the packet we received take appropriate action
  639:  switch(response->type) {
  640:   case PACKET_G:  //received guess data
  641: 
  642:       //store received data into received_guess[]
  643:       //lcd_clearscreen();
  644:      // lcd_printf("Received this guess: ");
  645:       for(i = 0; i < 4; i++){
  646:           received_guess[i] =  response->body.g.guess[i];
  647:           //io_printf(IO_HYPERTERM, "%c", received_guess[i]);
  648:      //     lcd_printf("%c", received_guess[i]);
  649:       }
  650: 
  651:      // First count the Bulls
  652:       for(i = 0; i < 4; i++){
  653:           if(received_guess[i] == number[i]){
  654:             bulls++;
  655:             // Invalidate found value to avoid double counting when looking for cows
  656:             received_guess[i] = 15;
  657:             }
  658:       }
  659: 
  660:       // Then count the cows
  661:       for(i = 0; i < 4; i++)
  662:         for(k = 0; k < 4; k++)
  663:           if(received_guess[k] == number[i])
  664:             cows++;
  665: 
  666:       // At this point, cows and bulls are up to date
  667:       if(bulls == 4){
  668:           lcd_clearscreen();
  669:           lcd_printf("You Lose!!!");
```

```
  670:            //io_printf(IO_HYPERTERM, "YOU LOSE!!\n\r");
  671:            *current_state = GAME_OVER;
  672:            timer_wait_seconds(3);
  673:          }
  674:
  675:        /*lcd_goto(24);
  676:        lcd_printf("Your # is %c%c%c%c (%dB, %dC)", number[0], number[1], number[2], numb
er[3], bulls, cows);
  677:        */
  678:        //send acknoledgment with cows and bulls
  679:        packet_create_y(&y_ack, bulls, cows);
  680:        cc2500_transmit_packet(&y_ack);
  681:        break;
  682:
  683:
  684:    case PACKET_Y:  //we received info about cows and bulls. display it and keep going if
 game did not end
  685:        gotAnswer = TRUE;
  686:        bulls = response->body.y.num_bulls;
  687:        cows = response->body.y.num_cows;
  688:
  689:        oldGuessesResults[oldGuessesPointer][0] = bulls;
  690:        oldGuessesResults[oldGuessesPointer][1] = cows;
  691:
  692:        lcd_clearscreen();
  693:        lcd_printf("Your guess #%d: ", oldGuessesPointer, key[0], key[1], key[2], key[3])
;
  694:        lcd_goto(24);
  695:        //io_printf(IO_HYPERTERM, "\n\rBulls: %d\tCows: %d\n\r\n", bulls, cows);
  696:        lcd_printf("Bulls: %d Cows: %d", bulls, cows);
  697:        timer_wait_seconds(2);
  698:
  699:            if(bulls == 4){
  700:                lcd_clearscreen();
  701:                lcd_printf("You WIN!!!!");
  702:                //io_printf(IO_HYPERTERM, "YOU WIN!!\n\r");
  703:                *current_state = GAME_OVER;
  704:                timer_wait_seconds(3);
  705:              }
  706:            else{
  707:              lcd_clearscreen();
  708:              lcd_printf("Guess:");
  709:              lcd_goto(24);
  710:            }
  711:
  712:        break;
  713:
  714:
  715:      case PACKET_SERVER_MESSAGE:
  716:        if(response->body.s.t == 'T') {
  717:          lcd_printf("Goodbye!");
  718:          timer_wait_seconds(2);
  719:          *current_state = GAME_OVER;
  720:        }
  721:        else {
  722:          //io_printf(IO_HYPERTERM, "Server Message of type '%c'\r\n=================\r\n
%s\r\n", response->body.s.t, response->body.s.message);
  723:        }
  724:        break;
  725:   }
  726:
  727: }
  728:
```

```
729:
730:
731:
```

```
 1:
 2: #include <stdio.h>
 3: #include <stdarg.h>
 4: #include "common.h"
 5: #include "hyperterm.h"
 6: #include "uart.h"
 7:
 8: #define HYPERTERM_UART  USE_UART0
 9:
10:
11: int hyperterm_echo = TRUE;
12:
13:
14: void hyperterm_init(void) {
15:   uart_switch(HYPERTERM_UART);
16:   uart_init();
17: }
18:
19: void hyperterm_set_echo(int stat){
20:   hyperterm_echo = stat;
21: }
22:
23: void hyperterm_putchar(char c) {
24:   uart_switch(HYPERTERM_UART);
25:   uart_put_byte(c);
26: }
27:
28: char hyperterm_getchar(void){
29:   char byte;
30:   uart_switch(HYPERTERM_UART);
31:   byte = uart_get_byte();
32:   if(hyperterm_echo) {
33:     hyperterm_putchar(byte);
34:   }
35:   return byte;
36: }
37:
38: char hyperterm_timeoutgetchar(void){
39:   char byte;
40:   uart_switch(HYPERTERM_UART);
41:   byte = uart_get_byte_timeout();
42:   if(hyperterm_echo && byte != 0x11) {
43:     hyperterm_putchar(byte);
44:   }
45:   return byte;
46: }
47:
48: void hyperterm_clear(void) {
49:     hyperterm_putchar(0xc);
50:     //hyperterm_putchar("\x1B[2J");
51: }
52:
```

```
    1: 
    2: #include <stdarg.h>
    3: #include <stdio.h>
    4: 
    5: #include "io.h"
    6: #include "keyboard.h"
    7: #include "hyperterm.h"
    8: #include "lcd.h"
    9: 
   10: #define IO_FORMAT_BUFFER 15
   11: #define IO_SUBFORMAT_BUFFER 32
   12: 
   13: void io_init(void){
   14:   keyboard_init();
   15:   hyperterm_init();
   16:   lcd_init();
   17: 
   18: }
   19: void io_putchar(int type, char c) {
   20:   if(type&IO_HYPERTERM) {
   21:     hyperterm_putchar(c);
   22:   }
   23:   if(type&IO_LCD) {
   24:     lcd_putchar(c);
   25:   }
   26: }
   27: 
   28: 
   29: int io_getchar(int type)  {
   30:   int ret = -1;
   31:   switch(type) {
   32:     case IO_KEYBOARD:
   33:       ret = keyboard_getchar();
   34:       break;
   35:     case IO_HYPERTERM:
   36:       ret = hyperterm_getchar();
   37:       break;
   38:     case IO_LCD:
   39:       break;
   40:   }
   41:   return ret;
   42: }
   43: 
   44: int io_timeoutgetchar(int type)  {
   45:   int ret = -1;
   46:   switch(type) {
   47:     case IO_KEYBOARD:
   48:       ret = keyboard_timeoutgetchar();
   49:       break;
   50:     case IO_HYPERTERM:
   51:       ret = hyperterm_timeoutgetchar();
   52:       break;
   53:     case IO_LCD:
   54:       break;
   55:   }
   56:   return ret;
   57: }
   58: 
   59: 
   60: void io_clear(int type) {
   61:   switch(type) {
   62:     case IO_KEYBOARD:
   63:       break;
```

```
   64:     case IO_HYPERTERM:
   65:       hyperterm_clear();
   66:       break;
   67:     case IO_LCD:
   68:       lcd_clearscreen();
   69:       break;
   70:   }
   71: }
   72:
   73: void io_printf(int out_type, char* format, ...) {
   74:
   75:   char buffer[IO_FORMAT_BUFFER];
   76:   char sub_format[IO_SUBFORMAT_BUFFER];
   77:   char type;
   78:   int j = 0;
   79:   int i, arg_size;
   80:   va_list args;
   81:   va_start(args, format);
   82:
   83:   for(j = 0; format[j] != '\0'; j++) {
   84:
   85:     if(format[j] == '%' ) {
   86:       arg_size = 0;
   87:       for(i = 0; i < IO_SUBFORMAT_BUFFER-1; i++) {
   88:         if((format[j+i] >= '0' && format[j+i] <= '9') || format[i] == '.' || format[j+i
] == '%') {
   89:           arg_size++;
   90:         }
   91:         else break;
   92:       }
   93:
   94:       for(i = 0; i <= arg_size; i++) {
   95:         sub_format[i] = format[j+i];
   96:       }
   97:       type = sub_format[i-1];
   98:       sub_format[i] = '\0';
   99:       j += arg_size;
  100:
  101:
  102:       switch(type) {
  103:         case '%':
  104:           buffer[0] = '%';
  105:           buffer[1] = '\0';
  106:           break;
  107:         case 'i':
  108:         case 'd':
  109:           sprintf(buffer, sub_format, va_arg(args, int));
  110:           break;
  111:         case 'c':
  112:           sprintf(buffer, sub_format, va_arg(args, int));
  113:           break;
  114:         case 'x':
  115:         case 'X':
  116:           sprintf(buffer, sub_format, va_arg(args, int));
  117:           for(i = 0; buffer[i] != '\0'; i++){
  118:             buffer[i] = buffer[i]==' '?'0':buffer[i];
  119:           }
  120:           break;
  121:         case 's':
  122:           sprintf(buffer, sub_format, va_arg(args, char*));
  123:
  124:           break;
  125:       }
```

```
126:
127:       for(i = 0; i < IO_FORMAT_BUFFER && buffer[i] != '\0'; i++) {
128:         io_putchar(out_type, buffer[i]);
129:       }
130:     }
131:     else {
132:       io_putchar(out_type, format[j]);
133:     }
134:   }
135:
136:   va_end(args);
137: }
138:
139:
140:
141: int io_gethex(int type) {
142:
143:
144:
145: }
146: //
147: //int convInt(char ascii) {
148: //  if(ascii >= '0' && ascii <= '9') return ascii-'0';
149: //  else return ascii-'A'+0xA;
150: //
151: //}
152: //int getHex(void) {
153: //  char c;
154: //  int val = 0;
155: //  c = hyperterm_getchar();
156: //  if(c == 0x1B) return -1;
157: //  val += convInt(c) << 4;
158: //  c = hyperterm_getchar();
159: //  if(c == 0x1B) return -1;
160: //  val += convInt(c);
161: //
162: //  return val;
163: //}
164:
```

```
    1:
    2: #include "keyboard.h"
    3: #include "common.h"
    4: #include "uart.h"
    5: #include "hyperterm.h"
    6: #include "timer.h"
    7:
    8: #define PORT2_KEYBOARD_INTERRUPT 0x01
    9: #define PORT5_SPI_REDIRECT 0x40
   10:
   11: #define KEYBOARD_SPI_PORT (USE_UART1 | USE_SPI)
   12:
   13: int keyboard_waiting = FALSE;
   14:
   15: int keyboard_shift_pressed = FALSE;
   16:
   17: char keyboard_irq = FALSE;
   18:
   19: char hexdisplay_value = 0x00;
   20:
   21:
   22: void hexdisplay_set(char b) {
   23:   hexdisplay_value = b;
   24:
   25:
   26:   uart_switch(KEYBOARD_SPI_PORT);
   27:   P5OUT |= PORT5_SPI_REDIRECT;
   28:
   29:   uart_spi_byte(hexdisplay_value, SPI_SINGLE_TRANS);
   30:
   31:   P5OUT &= ~PORT5_SPI_REDIRECT;
   32: }
   33:
   34: void keyboard_init(void) {
   35:
   36:   uart_switch(KEYBOARD_SPI_PORT);
   37:   uart_init();
   38:
   39:   P5DIR |= PORT5_SPI_REDIRECT;
   40:   P5OUT &= ~PORT5_SPI_REDIRECT; // Set standard to receiver
   41:
   42:   P2DIR &= ~PORT2_KEYBOARD_INTERRUPT; // input port
   43:   P2IES &= ~PORT2_KEYBOARD_INTERRUPT;   //Set Low to High interrupts
   44:
   45:   P2IE  |= PORT2_KEYBOARD_INTERRUPT;  //enable interrupts
   46: }
   47:
   48: char keyboard_getchar(void){
   49:   char scancode[3];
   50:   int make_code;
   51:   int key;
   52:
   53:   do {
   54:     keyboard_wait_scancode(scancode);
   55:     make_code = keyboard_make_code(scancode);
   56:     //hyperterm_printf("%4X\r\n", make_code);
   57:     key = keyboard_make_code_to_ascii(make_code);
   58:     //hyperterm_printf("%i %i\r\n", key, keyboard_scancode_release(scancode));
   59:   } while(keyboard_scancode_release(scancode) || key == -1);
   60:
   61:   //hyperterm_printf("%4X\r\n", make_code);
   62:   return key;
   63: }
```

```
 64:
 65: char keyboard_timeoutgetchar(void) {
 66:   char scancode[3];
 67:   int make_code;
 68:   int key;
 69:
 70:   do {
 71:     keyboard_waiting = TRUE;
 72:     keyboard_irq = FALSE;
 73:     timer_wait(10000);
 74:     if(!keyboard_irq) return 0x11;
 75:
 76:     keyboard_receive_scancode(scancode);
 77:     keyboard_waiting = FALSE;
 78:
 79:     make_code = keyboard_make_code(scancode);
 80:     //hyperterm_printf("%4X\r\n", make_code);
 81:     key = keyboard_make_code_to_ascii(make_code);
 82:     //hyperterm_printf("%i %i\r\n", key, keyboard_scancode_release(scancode));
 83:   } while(keyboard_scancode_release(scancode) || key == -1);
 84:
 85:   //hyperterm_printf("%4X\r\n", make_code);
 86:   return key;
 87:
 88: }
 89:
 90:
 91: int keyboard_make_code(char* scancode) {
 92:   int make_code = 0;
 93:   make_code |= scancode[2];
 94:   if(keyboard_scancode_release(scancode)) {
 95:     make_code |= (scancode[0] << 8);
 96:   }
 97:   else {
 98:     make_code |= (scancode[1] << 8);
 99:   }
100:   return make_code;
101: }
102:
103:
104: int keyboard_scancode_release(char* code) {
105:   if(code[1] == 0xF0) return TRUE;
106:   else return FALSE;
107: }
108:
109: void keyboard_wait_scancode(char* code) {
110:   keyboard_waiting = TRUE;
111:   _BIS_SR(CPUOFF);
112:   keyboard_receive_scancode(code);
113:   keyboard_waiting = FALSE;
114:   //hyperterm_printf("%2X %2X %2X\r\n", code[0], code[1], code[2]);
115: }
116:
117: void keyboard_receive_scancode(char* code) {
118:
119:   P5OUT |= PORT5_SPI_REDIRECT;
120:
121:   uart_switch(KEYBOARD_SPI_PORT);
122:   code[0] = uart_spi_byte(hexdisplay_value, SPI_START_TRANS);
123:   code[1] = uart_spi_byte(hexdisplay_value, SPI_CONT_TRANS);
124:   code[2] = uart_spi_byte(hexdisplay_value, SPI_END_TRANS);
125:
126:   P5OUT &= ~PORT5_SPI_REDIRECT;
```

```
127: }
128:
129: void port2_irq(void) __interrupt[PORT2_VECTOR] {
130:   char scancode[3];
131:   int make_code;
132:
133:   P2IFG = 0x0;                          //clearing the interrupt flag
134:   if(keyboard_waiting) {
135:     keyboard_irq = TRUE;
136:     _BIC_SR_IRQ(CPUOFF);
137:   }
138:   else {
139:     return;
140:     // Ignoring handling shift for now, due to issues
141:     keyboard_receive_scancode(scancode);
142:     make_code = keyboard_make_code(scancode);
143:     if(make_code == 0x59 || make_code == 0x12) {
144:       keyboard_shift_pressed = !keyboard_scancode_release(scancode);
145:     }
146:   }
147: }
148:
149:
150:
151:
152: //key codes from Scancode Set 2 here: http://www.quadibloc.com/comp/scan.htm
153: int keyboard_make_code_to_ascii(int make_code) {
154:   int c;
155:
156:   switch(make_code) {
157:     case 0x001C: return 'a';
158:     case 0x0032: return 'b';
159:     case 0x0021: return 'c';
160:     case 0x0023: return 'd';
161:     case 0x0024: return 'e';
162:     case 0x002B: return 'f';
163:     case 0x0034: return 'g';
164:     case 0x0033: return 'h';
165:     case 0x0043: return 'i';
166:     case 0x003B: return 'j';
167:     case 0x0042: return 'k';
168:     case 0x004B: return 'l';
169:     case 0x003A: return 'm';
170:     case 0x0031: return 'n';
171:     case 0x0044: return 'o';
172:     case 0x004D: return 'p';
173:     case 0x0015: return 'q';
174:     case 0x002D: return 'r';
175:     case 0x001B: return 's';
176:     case 0x002C: return 't';
177:     case 0x003C: return 'u';
178:     case 0x002A: return 'v';
179:     case 0x001D: return 'w';
180:     case 0x0022: return 'x';
181:     case 0x0035: return 'y';
182:     case 0x001A: return 'z';
183:     case 0x0066: return '\b'; //backspace
184:     case 0x0029: return ' ';
185:     case 0x000D: return '\t';
186:     case 0x005A:
187:     case 0xE05A:
188:       return '\r';
189:     case 0x0076: return 0x1B; //Esc
```

```
190:
191:      case 0x0045:
192:      case 0x0070:
193:        return '0';
194:      case 0x0016:
195:      case 0x0069:
196:        return '1';
197:      case 0x001E:
198:      case 0x0072:
199:        return '2';
200:      case 0x0026:
201:      case 0x007A:
202:        return '3';
203:      case 0x0025:
204:      case 0x006B:
205:        return '4';
206:      case 0x002E:
207:      case 0x0073:
208:        return '5';
209:      case 0x0036:
210:      case 0x0074:
211:        return '6';
212:      case 0x003D:
213:      case 0x006C:
214:        return '7';
215:      case 0x003E:
216:      case 0x0075:
217:        return '8';
218:      case 0x0046:
219:      case 0x007D:
220:        return '9';
221:      case 0x004E: return '-';
222:      case 0x0055: return '=';
223:      case 0x005D: return '\\';
224:      case 0x004A: return '/';
225:
226:    }
227:    return -1;
228: }
229:
```

```
  1: #include <stdio.h>
  2: #include <stdarg.h>
  3: #include "common.h"
  4: #include "timer.h"
  5: #include "lcd.h"
  6: #include "wireless.h"
  7:
  8: #define LCD_FORMAT_BUFFER 15
  9: #define LCD_SUBFORMAT_BUFFER 7
 10:
 11: #define LCD_DL 0 //4-bit mode
 12: #define LCD_N  1 //2-line mode
 13: #define LCD_F  1 //display ON
 14: #define LCD_D  1 //display ON
 15: #define LCD_C  0 //cursor OFF
 16: #define LCD_B  0 //cursor blink ON
 17: #define LCD_SH 0 //Entire Shift on
 18: #define LCD_ID 1 //increment ON
 19:
 20: #define LCD_E   0x40          //Enable is plugged in P6.6
 21: #define LCD_RS  0x20         //Register Select is plugged in P6.5
 22: #define LCD_RW  0x10          // Read/Write plugged in P6.4
 23:
 24: #define LCD_DB4_0   0x01      //DB4/DB0 is plugged in P6.0
 25: #define LCD_DB5_1   0x02     //DB5/DB1 is plugged in P6.1
 26: #define LCD_DB6_2   0x04      //DB6/DB2 is plugged in P6.2
 27: #define LCD_DB7_3   0x08      //DB7/DB3 is plugged in P6.3
 28:
 29: #define LINE_1_ADDRESS 0x00
 30: #define LINE_2_ADDRESS 0x40
 31:
 32: char currentCursor;
 33:
 34: void lcd_init(void){
 35:   //char status = 0x80;
 36:   timer_wait(2000); //After power up wait 40ms
 37:
 38:   P6DIR = 0xFF; //everybody OUT !
 39:   P6SEL = 0x00; //select GPIO mode for everything
 40:
 41:   P6OUT = 0x00;
 42:
 43:   timer_wait(4800);
 44:
 45:
 46:   P6OUT = LCD_DB5_1 + LCD_DB4_0;
 47:   lcd_pulse_e();
 48:   P6OUT = LCD_DB5_1 + LCD_DB4_0;
 49:   lcd_pulse_e();
 50:   P6OUT = LCD_DB5_1 + LCD_DB4_0;
 51:   lcd_pulse_e();
 52:
 53:   /*Function Set*/
 54:   P6OUT = LCD_DB5_1;
 55:   lcd_pulse_e();
 56:
 57:   P6OUT = LCD_DB5_1;
 58:   lcd_pulse_e();
 59:
 60:   P6OUT = LCD_DB7_3*LCD_N + LCD_DB6_2*LCD_F;
 61:   lcd_pulse_e();
 62:
 63:   timer_wait(480);  //wait 600us
```

```
 64:
 65:   /*Display ON/OFF Control*/
 66:   P6OUT = 0x00;
 67:   lcd_pulse_e();
 68:   P6OUT = LCD_DB7_3 + LCD_DB6_2*LCD_D + LCD_DB5_1*LCD_C + LCD_DB4_0*LCD_B;
 69:   lcd_pulse_e();
 70:
 71:  timer_wait(480);
 72:
 73:   /*Display Clear*/
 74:   P6OUT = 0x00;
 75:   lcd_pulse_e();
 76:   //timer_wait(480);  //wait 600us
 77:   P6OUT = LCD_DB4_0;
 78:   lcd_pulse_e();
 79:
 80:  timer_wait(480);
 81:
 82:   /*Entry Mode Set*/
 83:   P6OUT = 0x00;
 84:   lcd_pulse_e();
 85:   P6OUT = LCD_DB6_2 + LCD_DB5_1*LCD_ID + LCD_DB4_0*LCD_SH;
 86:   lcd_pulse_e();
 87:
 88:   timer_wait(480);//Initialization End
 89:
 90:   currentCursor = 0;
 91:   //P6OUT =0x00;
 92:   //lcd_pulse_e();
 93:
 94:   //lcd_read_status();
 95:   /*while(status&0x80 != 0){
 96:   status = lcd_read_status();
 97:   } */
 98:
 99:   //lcd_writebyte('A');
100:
101: }
102:
103: void lcd_writebyte(char b ){
104:   char status = 0x80;
105:   P6DIR = 0xFF;
106:   P6OUT = 0x00;
107:   P6OUT &= ~LCD_RS;
108:   P6OUT &= ~LCD_RW;
109:
110:   P6OUT = LCD_DB7_3;
111:   lcd_pulse_e();
112:
113:   P6OUT = 0x00;
114:   lcd_pulse_e();
115:
116:   P6OUT |= LCD_RS;
117:   P6OUT &= ~LCD_RW;
118:   P6OUT &= ~0x0F;
119:
120:   P6OUT |= (b >> 4)&0x0F; // first four bits are db7,db6,db5,db4
121:   lcd_pulse_e();
122:
123:   P6OUT &= ~0x0F;
124:
125:   P6OUT |= b & 0x0F;
126:   lcd_pulse_e();
```

```
127:
128:
129:   //while(status&0x80 != 0){
130:   //status = lcd_read_status();
131:   //}
132: }
133:
134: char lcd_read_status(void){
135:   char rb = 0;
136:   P6DIR = LCD_RS + LCD_RW + LCD_E; //only those three are out values, rest is IN
137:   P6OUT = LCD_RW; //set read pin high
138:
139:   lcd_pulse_e();
140:   rb += (P6IN&0x0F) << 4;
141:
142:   lcd_pulse_e();
143:   rb += (P6IN&0x0F);
144:
145:   P6DIR = 0xFF; //put it back to ALL OUT
146:   return rb;
147: }
148:
149:
150: void lcd_clearscreen(void){
151:   P6OUT &= ~LCD_RS;
152:   P6OUT &= ~LCD_RW;
153:   send_data(0x01);
154:   currentCursor = 0;
155:   timer_wait(500);
156: }
157:
158: void lcd_pulse_e(void)
159: {
160:   wireless_receive_status(FALSE);
161:   //set e high
162:   P6OUT |= LCD_E;
163:
164:   //pause
165:   timer_wait(3);
166:
167:   //set e low
168:   P6OUT &= ~LCD_E;
169:
170:   //pause
171:   timer_wait(3);
172:   wireless_receive_status(TRUE);
173: }
174:
175: void send_data(unsigned char data)
176: {
177:   //send high part first
178:   send_nibble((data>>4) & 0x0F);
179:   send_nibble(data & 0x0F);
180: }
181:
182: void send_nibble(unsigned char data)
183: {
184:   //convert the nibble to the appropriate pins
185:   if(data & 0x01)
186:     P6OUT |= LCD_DB4_0;
187:   else
188:     P6OUT &= ~LCD_DB4_0;
189:
```

```
190:   if(data & 0x02)
191:     P6OUT |= LCD_DB5_1;
192:   else
193:     P6OUT &= ~LCD_DB5_1;
194:
195:   if(data & 0x04)
196:     P6OUT |= LCD_DB6_2;
197:   else
198:     P6OUT &= ~LCD_DB6_2;
199:
200:   if(data & 0x08)
201:     P6OUT |= LCD_DB7_3;
202:   else
203:     P6OUT &= ~LCD_DB7_3;
204:
205:   lcd_pulse_e();
206: }
207:
208:
209: void lcd_goto(char p){
210:   P6OUT &= ~LCD_RS;
211:   P6OUT &= ~LCD_RW;
212:   if (p < 24 ) send_data(0x80 | (LINE_1_ADDRESS + p));
213:   else if (p < 48 ) send_data(0x80 | (LINE_2_ADDRESS + (p-24)));
214:   currentCursor = p;
215: }
216:
217:
218: void lcd_putchar(char c){
219:   lcd_goto(currentCursor);
220:   currentCursor++;
221:   P6OUT |= LCD_RS;
222:   P6OUT &= ~LCD_RW;
223:   send_data(c);
224: }
225: void printString(char* string)
226: {
227:   while ((*string) != '\0'){
228:     lcd_putchar(*string);
229:     string++;
230:   }
231: }
232:
233: char lcd_printf(char* format, ...) {
234:   char buffer[LCD_FORMAT_BUFFER];
235:   char sub_format[LCD_SUBFORMAT_BUFFER];
236:   char type;
237:   int j = 0;
238:   int i, arg_size;
239:   va_list args;
240:   va_start(args, format);
241:
242:   for(j = 0; format[j] != '\0'; j++) {
243:
244:     if(format[j] == '%' ) {
245:       arg_size = 0;
246:       for(i = 0; i < LCD_SUBFORMAT_BUFFER-1; i++) {
247:         if((format[j+i] >= '0' && format[j+i] <= '9') || format[i] == '.' || format[j+i
] == '%') {
248:           arg_size++;
249:         }
250:         else break;
251:       }
```

```
252:
253:       for(i = 0; i <= arg_size; i++) {
254:         sub_format[i] = format[j+i];
255:       }
256:       type = sub_format[i-1];
257:       sub_format[i] = '\0';
258:       j += arg_size;
259:
260:
261:       switch(type) {
262:         case '%':
263:           buffer[0] = '%';
264:           buffer[1] = '\0';
265:           break;
266:         case 'i':
267:         case 'd':
268:           sprintf(buffer, sub_format, va_arg(args, int));
269:           break;
270:         case 'c':
271:           sprintf(buffer, sub_format, va_arg(args, int));
272:           break;
273:         case 'x':
274:         case 'X':
275:           sprintf(buffer, sub_format, va_arg(args, int));
276:           for(i = 0; buffer[i] != '\0'; i++){
277:             buffer[i] = buffer[i]==' '?'0':buffer[i];
278:           }
279:           break;
280:         case 's':
281:           sprintf(buffer, sub_format, va_arg(args, char*));
282:
283:           break;
284:       }
285:
286:       for(i = 0; i < LCD_FORMAT_BUFFER && buffer[i] != '\0'; i++) {
287:         lcd_putchar(buffer[i]);
288:       }
289:     }
290:     else {
291:       lcd_putchar(format[j]);
292:     }
293:   }
294:
295:   va_end(args);
296: }
297:
298: int lcd_checkAlphabet(char key){
299:   if (key >= '0' && key <= '9') return TRUE;
300:   if (key >= 'A' && key <= 'Z') return TRUE;
301:   if (key >= 'a' && key <= 'z') return TRUE;
302:   return FALSE;
303: }
304:
305: void lcd_changeline(void){
306:   if (currentCursor >23) lcd_goto(LCD_LINE1);
307:   else lcd_goto(LCD_LINE2);
308: }
309:
310:
311: void lcd_clearline(int line){
312:   if(line == 1){
313:     lcd_goto(LCD_LINE1);
314:     lcd_printf("                        ");
```

```
 315:     lcd_goto(LCD_LINE1);
 316: 
 317:   }
 318:   if(line == 2){
 319:     lcd_goto(LCD_LINE2);
 320:     lcd_printf("                        ");
 321:     lcd_goto(LCD_LINE2);
 322:   }
 323: 
 324: 
 325: }
 326: 
```

```
  1:
  2: #include "common.h"
  3: #include "keypad.h"
  4: #include "timer.h"
  5: #include "cc2500.h"
  6: #include "io.h"
  7: #include "keyboard.h"
  8: #include "lcd.h"
  9: #include "game.h"
 10: //#include "wifiProtocol.h"
 11: #include "packet.h"
 12: #include "flash.h"
 13: #include "wireless.h"
 14:
 15:
 16:
 17: int convInt(char ascii) {
 18:   if(ascii >= '0' && ascii <= '9') return ascii-'0';
 19:   else return ascii-'A'+0xA;
 20:
 21: }
 22: int getHex(void) {
 23:   char c;
 24:   int val = 0;
 25:   c = io_getchar(IO_HYPERTERM);
 26:   if(c == 0x1B) return -1;
 27:   val += convInt(c) << 4;
 28:   c = io_getchar(IO_HYPERTERM);
 29:   if(c == 0x1B) return -1;
 30:   val += convInt(c);
 31:
 32:   return val;
 33: }
 34:
 35:
 36: /**
 37:  * Loop over all registers and print address, name and current value
 38:  */
 39: void dump_registers(void) {
 40:   int i;
 41:   char data;
 42:   char* name = NULL;
 43:
 44:   io_printf(IO_HYPERTERM, "\r\n");
 45:
 46:   for(i = 0; i < 0xFF; i++) {
 47:     name = cc2500_reg_get_name(i);
 48:     if(name != NULL){
 49:       data = cc2500_read_reg(i);
 50:
 51:       io_printf(IO_HYPERTERM, "0x%2X : %s = 0x%2X\r\n", i, name, data);
 52:     }
 53:   }
 54:
 55:   data = cc2500_status();
 56:   io_printf(IO_HYPERTERM, "\r\nStatus: 0x%2X\r\n", data);
 57: }
 58:
 59:
 60: /**
 61:  * Request register address, display current value, and request new value
 62:  */
 63: void modify_registers(void) {
```

```
 64:   int addr;
 65:   int val;
 66:   char* name;
 67:
 68:   io_printf(IO_HYPERTERM, "What register would you like to modify?\r\n: 0x");
 69:   addr = getHex();
 70:   if(addr == -1){
 71:     io_printf(IO_HYPERTERM, "\r\n");
 72:     return;
 73:   }
 74:   name = cc2500_reg_get_name(addr);
 75:
 76:   if(name == NULL) {
 77:     io_printf(IO_HYPERTERM, "\r\nInvalid Register Address\r\n");
 78:   }
 79:   else {
 80:
 81:     val = cc2500_read_reg(addr);
 82:
 83:     io_printf(IO_HYPERTERM, "Register Modification for %s(0x%2X):\r\nValue = 0x%2X chan
ged to => 0x", name, addr, val);
 84:
 85:     val = getHex();
 86:     if(val == -1){
 87:       io_printf(IO_HYPERTERM, "\r\n");
 88:       return;
 89:     }
 90:     cc2500_write_reg(addr, val);
 91:
 92:   }
 93: }
 94:
 95:
 96: /**
 97:  * Read data from the RXBUF of the CC2500 and dump to terminal
 98:  */
 99: void read_data(void) {
100:   char val;
101:   int i;
102:   int size = 0;
103:
104:   packet r;
105:   if (wireless_packet_available()){
106:     if(wireless_packet_receive(&r)) {
107:       packet_print(IO_HYPERTERM, &r);
108:     }
109:     else {
110:       io_printf(IO_HYPERTERM, "Error receiving packet, please try again\r\n");
111:     }
112:   }
113:   else {
114:     io_printf(IO_HYPERTERM,"No packets in queue\r\n");
115:   }
116: }
117:
118: /**
119:  * Transmit data then proceed as above
120:  */
121: void echo_data(void) {
122:   int i;
123:   char val;
124:   char* dat;
125:   char guess[] = {'1', '2', '3', '4'};
```

```
126:   packet r;
127:   packet p;
128:
129:   packet_create_h(&p, "Logan Smyth");
130:   cc2500_transmit_packet(&p);
131:
132:   for(i = 0; i < 5; i++) {
133:     if(wireless_packet_receive(&r)){
134:       io_printf(IO_HYPERTERM, "===============\r\n");
135:       packet_print(IO_HYPERTERM, &r);
136:
137:     }
138:     else {
139:       i--;
140:     }
141:   }
142: }
143:
144:
145:
146: #define STATE_MAINMENU_RESET '1'
147: #define STATE_MAINMENU_DUMP_REGS '2'
148: #define STATE_MAINMENU_MODIFY '3'
149: #define STATE_MAINMENU_READ_DATA '4'
150: #define STATE_MAINMENU_ECHO '5'
151: #define STATE_MAINMENU_EXIT '6'
152: #define STATE_MAINMENU_LISTEN '7'
153: #define STATE_MAINMENU_SEND '8'
154: #define STATE_PLAYGAME '9'
155: #define STATE_FLASH '0'
156: #define STATE_MAINMENU_KEYBOARD1 'a'
157: #define STATE_MAINMENU_KEYBOARD2 'A'
158: /**
159:  * Main menu state machine
160:  */
161: void main_menu(void) {
162:   int type, i;
163:   char guess[] = {'1', '2', '3', '4'};
164:   char data_for_flash;
165:   char* name = NULL;
166:
167:   packet r;
168:   packet p;
169:
170:   int state = FALSE;
171:   do {
172:     io_printf(IO_HYPERTERM, "\r\nChoose an Option:\r\n 1) Reset\r\n 2) Dump all Registe
rs\r\n 3) Modify Register\r\n 4) Read Data\r\n 5) Echo Data\r\n 6) Exit\r\n 7) Listen for 20 P
ackets\r\n 8) Send A Packet\r\n 9) Play Game \n\r 0) Write/Read Flash \n\r A) Keyboard Testing
\r\n");
173: //    state = io_getchar(IO_KEYBOARD);
174:     state = io_getchar(IO_HYPERTERM);
175:
176:
177:     switch (state) {
178:       case STATE_FLASH:
179:
180:       for(i = 0; i < 0xFF; i++) {
181:           data_for_flash = cc2500_read_reg(i);
182:           name = cc2500_reg_get_name(i);
183:           if(name != NULL){
184:              writeToFlashMemory(data_for_flash,i);
185:           }
```

```
  186:         }
  187:
  188:         io_printf(IO_HYPERTERM, "\r\n Register values saved to flash.\r\n Check Memory Wi
ndow starting at address 1000.\r\n");
  189:         io_printf(IO_HYPERTERM, "\r\n Now reading the values: \r\n");
  190:
  191:         for(i = 0; i < 0xFF; i++) {
  192:             data_for_flash =  readFromFlashMemory(i);
  193:             name = cc2500_reg_get_name(i);
  194:             if(name != NULL){
  195:                io_printf(IO_HYPERTERM, "0x%2X : %s = 0x%2X\r\n", i, name, data_for_flash)
;
  196:             }
  197:
  198:         }
  199:          break;
  200:         case STATE_MAINMENU_RESET:
  201:           cc2500_init();
  202:           break;
  203:         case STATE_MAINMENU_DUMP_REGS:
  204:           dump_registers();
  205:           break;
  206:         case STATE_MAINMENU_MODIFY:
  207:           modify_registers();
  208:           break;
  209:         case STATE_MAINMENU_READ_DATA:
  210:           read_data();
  211:           break;
  212:         case STATE_MAINMENU_ECHO:
  213:           echo_data();
  214:           break;
  215:         case STATE_MAINMENU_LISTEN:
  216:           for(i = 0; i < 20; i++) {
  217:             if(wireless_packet_receive(&r)){
  218:               io_printf(IO_HYPERTERM, "================\r\n");
  219:               packet_print(IO_HYPERTERM, &r);
  220:             }
  221:             else {
  222:               i -= 1;
  223:             }
  224:           }
  225:           break;
  226:         case STATE_MAINMENU_SEND:
  227:           io_printf(IO_HYPERTERM, "What type of packet would you like to send? \r\n");
  228:           type = io_getchar(IO_HYPERTERM);
  229:           i = TRUE;
  230:           switch(type){
  231:             case 'h':
  232:               packet_create_h(&p, "Logan Smyth");
  233:               break;
  234:             case 'r':
  235:               packet_create_r(&p, "Logan Smyth");
  236:               break;
  237:             case 'z':
  238:               packet_create_z(&p, 23);
  239:               break;
  240:             case 'g':
  241:               packet_create_g(&p, 12, guess);
  242:               break;
  243:             case 'y':
  244:               packet_create_y(&p, '2','3');
  245:               break;
  246:             case 's':
```

```
 247:             packet_create_s(&p, 'T',"HELLO THERE");
 248:             break;
 249:           default:
 250:             io_printf(IO_HYPERTERM, "Unknown Type\r\n");
 251:             i = FALSE;
 252:         }
 253:         if(i) {
 254:           cc2500_transmit_packet(&p);
 255:           packet_print(IO_HYPERTERM, &p);
 256:         }
 257:         break;
 258: 
 259:       case STATE_PLAYGAME:
 260:         playGame();
 261:         break;
 262:       case STATE_MAINMENU_KEYBOARD1:
 263:       case STATE_MAINMENU_KEYBOARD2:
 264:         while((data_for_flash = io_getchar(IO_KEYBOARD)) != 0x1b) io_printf(IO_HYPERTER
M, "%c",data_for_flash);
 265:         break;
 266:     }
 267:   } while(state != STATE_MAINMENU_EXIT);
 268: }
 269: 
 270: 
 271: void main(void) {
 272:   packet p;
 273:   char guess[] = {'1', '2', '3', '4'};
 274:   char temp;
 275: 
 276:   // Stop watchdog.
 277:   WDTCTL = WDTPW + WDTHOLD;
 278:   _EINT();
 279: 
 280:   // Turning 8MHZ Oscillator ON
 281:   BCSCTL1 &= ~XT2OFF;
 282: 
 283:   // Route XT2 into SMCLK and MCLK
 284:   BCSCTL2 |= SELM1|SELS;
 285:   //BCSCTL2 |= DIVM_3; //Divide 8MHZ by 4 = 2  MHz
 286: 
 287:   // ACLK (32.768KHz), clear TAR
 288:   TACTL = TASSEL_1 + TACLR + MC1;
 289: 
 290:   timer_init();
 291:   cc2500_init();
 292: 
 293: 
 294:   io_init();
 295:   wireless_init();
 296: 
 297:   cc2500_write_reg(CC2500REG_CHANNR , 0x07);
 298: 
 299:   if(!cc2500_verify()) {
 300:     io_printf(IO_HYPERTERM, "== You need to poke the CPLD ==");
 301:     while(!cc2500_verify()) {
 302:       timer_wait(10000);
 303:     }
 304:   }
 305: 
 306:   io_printf(IO_HYPERTERM,"== Init successful ==");
 307: 
 308: 
```

```
309:     hexdisplay_set(0xAF);
310:
311:     playGame();
312:     main_menu();
313:
314:     return;
315: }
316:
317:
```

```
    1:
    2: #include <stdarg.h>
    3: #include <string.h>
    4: #include <stdio.h>
    5: #include "packet.h"
    6: #include "io.h"
    7: #include "cc2500.h"
    8:
    9:
   10: char packet_src = 0x00;
   11: char packet_dest = 0x00;
   12:
   13: void packet_source(char src) {
   14:   packet_src = src;
   15:   cc2500_write_reg(CC2500REG_ADDR, src);
   16: }
   17: void packet_destination(char dest) {
   18:   packet_dest = dest;
   19: }
   20:
   21:
   22: void packet_print(int out_type, packet *p) {
   23:   int i;
   24:   char* temp;
   25:   char* dat = (char*) p;
   26:   io_printf(out_type, "Packet Data:\r\n");
   27:   io_printf(out_type, "Length: 0x%2X\r\n", p->len);
   28:   io_printf(out_type, "Dest: 0x%2X\r\n", p->dest);
   29:   io_printf(out_type, "Src: 0x%2X\r\n", p->src);
   30:   io_printf(out_type, "RSSI: 0x%2X\r\n", p->rssi);
   31:   io_printf(out_type, "LQI: 0x%2X\r\n", p->lqi);
   32:   switch(p->type) {
   33:     case PACKET_H:
   34:       io_printf(out_type, "Host Beacon from %s\r\n", p->body.h.name);
   35:       break;
   36:     case PACKET_R:
   37:       io_printf(out_type, "Join Request from %s\r\n", p->body.r.name);
   38:       break;
   39:     case PACKET_Z:
   40:       io_printf(out_type, "Heartbeat %i\r\n", p->body.z.num_beats);
   41:       break;
   42:     case PACKET_G:
   43:       io_printf(out_type, "Guess #%i of \"%c%c%c%c\"\r\n", p->body.g.guess_count, p->bo
dy.g.guess[0],p->body.g.guess[1],p->body.g.guess[2],p->body.g.guess[3]);
   44:       break;
   45:     case PACKET_Y:
   46:       io_printf(out_type, "Guess Response Bulls:%c Cows:%c\r\n",p->body.y.num_bulls,p->
body.y.num_cows);
   47:       break;
   48:     case PACKET_S:
   49:       temp = p->body.s.message;
   50:       io_printf(out_type, "Server Message Type %c saying \"%s\"\r\n", p->body.s.t, temp
);
   51:       break;
   52:
   53:     default:
   54:       io_printf(out_type, "UNKNOWN: \"");
   55:       for(i = 2; i <= p->len && i < sizeof(packet); i++){
   56:         io_printf(out_type, "%c", dat[i]);
   57:       }
   58:       io_printf(out_type, "\"\r\n");
   59:       break;
   60:   }
```

```
 61: }
 62:
 63: void packet_create_basic(packet* p) {
 64:   p->len = 3;
 65:   p->dest = packet_dest;
 66:   p->src = packet_src;
 67:   p->rssi = 0;
 68:   p->lqi = 0;
 69: }
 70:
 71: void packet_create_h(packet* p, char* name) {
 72:   packet_create_basic(p);
 73:   p->type = PACKET_H;
 74:   strncpy(p->body.h.name, name, 19);
 75:   p->body.h.name[19] = '\0';
 76:   p->len += (strlen(p->body.h.name)+1);
 77: }
 78:
 79: void packet_create_r(packet* p, char* name) {
 80:   packet_create_basic(p);
 81:   p->type = PACKET_R;
 82:   strncpy(p->body.r.name, name, 19);
 83:   p->body.r.name[19] = '\0';
 84:   p->len += (strlen(p->body.r.name)+1);
 85: }
 86:
 87: void packet_create_z(packet* p, char num_beats) {
 88:   packet_create_basic(p);
 89:   p->type = PACKET_Z;
 90:   p->body.z.num_beats = num_beats;
 91:   p->len += 1;
 92: }
 93:
 94: void packet_create_g(packet* p, int guess_count, char* guess) {
 95:   int i;
 96:   packet_create_basic(p);
 97:   p->type = PACKET_G;
 98:   p->body.g.guess_count = guess_count;
 99:   for(i = 0; i < 4; i++) {
100:     p->body.g.guess[i] = guess[i];
101:   }
102:   p->len += 2+4;
103: }
104:
105: void packet_create_y(packet* p, char bulls, char cows) {
106:   packet_create_basic(p);
107:   p->type = PACKET_Y;
108:   p->body.y.num_bulls = bulls;
109:   p->body.y.num_cows = cows;
110:   p->len += 2;
111: }
112:
113: void packet_create_s(packet* p, char type, char* mess) {
114:   packet_create_basic(p);
115:   p->type = PACKET_S;
116:
117:   p->body.s.t = type;
118:   strncpy(p->body.s.message, mess, 29);
119:   p->body.s.message[29] = '\0';
120:   p->len += 1+strlen(p->body.s.message)+1;
121: }
122:
```

```
    1:
    2: #include "common.h"
    3: #include "timer.h"
    4:
    5: int ticks = 0;
    6: int seconds = 0;
    7:
    8: char timer_sleeping = FALSE;
    9:
   10: char timer_waiting(void) {
   11:   return timer_sleeping;
   12: }
   13:
   14: void timer_init(void) {
   15:   timer_reset();
   16:
   17:   TACCR2 = TAR + 512;                    // Interrupt every 256/32768 = 1/128 = 0.007812
5 seconds
   18:   TACCTL2 |= CCIE;   // CCR2 interrupt enabled
   19:
   20: }
   21: void timer_reset(void){
   22:         seconds = 0;
   23:         ticks = 0;
   24: }
   25:
   26: /**
   27:  * Iterrupt handler for TimerA
   28:  *
   29:  */
   30: void timerA_wakeup (void) __interrupt[TIMERA1_VECTOR] {
   31:   int vector = TAIV;
   32:
   33:   if (vector == 0x2){   //from Wait function
   34:     _BIC_SR_IRQ(CPUOFF);
   35:   }
   36:   else if(vector == 0x4) {      //for get_ticks function
   37:     TACCR2 += 512;
   38:     ticks++;
   39:     if(ticks & 0x40) { // every (64*512/32768) = 1 second
   40:       seconds++;
   41:       ticks = 0;
   42:     }
   43:     if(timer_wake_needed()) _BIC_SR_IRQ(CPUOFF);
   44:   }
   45: }
   46:
   47: /**
   48:  * Wait a given number of clock ticks by using TimerA
   49:  *
   50:  */
   51: void timer_wait(int t){
   52:
   53:   TACCR1 = TAR + t;
   54:   TACCTL1 |= CCIE;      // enable timer A1
   55:   timer_sleeping = TRUE;
   56:   _BIS_SR(CPUOFF);
   57:   timer_sleeping = FALSE;
   58:
   59:   TACCTL1 &= ~CCIE;     //disable timer A1
   60:
   61: }
   62:
```

```
  63: void timer_wait_seconds(int seconds){
  64:   while(seconds-- > 0)
  65:     timer_wait(32767);
  66:
  67: }
  68:
  69: int timer_get_ticks(void) {
  70:         return ticks;
  71: }
  72: int timer_get_seconds(void) {
  73:         return seconds;
  74: }
  75:
  76:
  77: char timer_wake_needed(void) {
  78:   return (timer_sleeping && TAIV == 0x02 && TACTL&TAIFG);
  79: }
  80:
```

```
  1:
  2: #include "common.h"
  3: #include "uart.h"
  4: #include "timer.h"
  5:
  6: int uart_port = USE_UART0;
  7: int uart_status = FALSE;
  8: int newbyte = FALSE;
  9: char uart_rx_buffer = 0x0;
 10:
 11: char transaction_sr;
 12:
 13: // Comment in H file
 14: int uart_switch(int type) {
 15:   int old_type = uart_port;
 16:   uart_port = type;
 17:   return old_type;
 18: }
 19:
 20: void uart_config_io(void) {
 21:   if(uart_port & USE_UART0) {
 22:     if(uart_port & USE_SPI) {
 23:       P3SEL |= 0x0E;    // All use Prt
 24:       P3DIR |= 0x0B;    // P3.0,1,3 OUT
 25:       P3OUT |= 0x01; // STE is high.
 26:     }
 27:     else {
 28:       P3SEL |= 0x30;  // P3.4,5 = USART0 TXD/RXD
 29:       P3DIR |= 0x10;  // P3.4 output direction
 30:     }
 31:   }
 32:   else if(uart_port & USE_UART1) {
 33:     if(uart_port & USE_SPI) {
 34:       P5SEL |= 0x0E;    // Set all to use port and STE to be GPIO
 35:       P5DIR |= 0x0B;    //P5.0,1,3 OUT
 36:       P5OUT |= 0x01;    // STE is high.
 37:     }
 38:     else {
 39:       P3SEL |= 0xC0;                        // P3.6,7 = USART1 option select
 40:       P3DIR |= 0x20;                        // P3.6 = output direction
 41:     }
 42:   }
 43:
 44: }
 45:
 46: // Comment in H file
 47: void uart_init(void) {
 48:   unsigned int i;
 49:   int ctl = 0, tctl = 0, br0, br1, mctl;
 50:
 51:   do {
 52:     IFG1 &= ~OFIFG;                       // Clear OSCFault flag
 53:     for (i = 0xFF; i > 0; i--);           // Time for flag to set
 54:   } while ((IFG1 & OFIFG) != 0);         // OSCFault flag still set?
 55:
 56:   if(uart_port & USE_SPI) {
 57:     ctl |= SWRST;       //SPI
 58:     tctl |=  CKPH | STC;
 59:     ctl |= SYNC | MM; // SPI
 60:   }
 61:
 62:   //Select SMCLK clk
 63:   tctl |= SSEL1|SSEL0;
```

```
   64:
   65:   // 8-bit characters
   66:   ctl |= CHAR;
   67:
   68:   // Set baud rate 67.5 kbps @ 8MHz
   69:   // Calculated here: http://www.daycounter.com/Calculators/MSP430-Uart-Calculator.phtm
l
   70:   /* uart0 8000000Hz 57595bps =~ 8000000/(0x8A.0x00) */
   71:   br0=0x8A; //setting the baudrate divider
   72:   br1=0x00;
   73:   mctl=0xEF;
   74:
   75:   uart_config_io();
   76:
   77:   if(uart_port & USE_UART0) {
   78:     U0CTL = ctl;
   79:     U0BR0 = br0;
   80:     U0BR1 = br1;
   81:     U0MCTL = mctl;
   82:     U0TCTL = tctl;
   83:
   84:     //Enabling the UART0 TX/RX Module
   85:     ME1 |= UTXE0 + URXE0;
   86:     IE1 |= URXIE0;                        // Enable USART0 RX interrupt
   87:
   88:     U0CTL &= ~SWRST;
   89:
   90:   }
   91:   else if(uart_port & USE_UART1) {
   92:     // 9600 baud
   93: //    br0 = 0x3;
   94: //    br1 = 0x41;
   95: //    mctl = 0x92;
   96:
   97:
   98:     U1CTL = ctl;
   99:     U1BR0 = br0;
  100:     U1BR1 = br1;
  101:     U1MCTL = mctl;
  102:     U1TCTL = tctl;
  103:
  104:     //Enabling the UART1 TX/RX Module
  105:     //ME2 |= UTXE1 + URXE1;
  106:     ME2 |= USPIE1;
  107:     IE2 |= URXIE1;                        // Enable USART1 RX interrupt
  108:
  109:     U1CTL &= ~SWRST;
  110:   }
  111:
  112: }
  113:
  114: // Comment in H file
  115: void uart_put_byte(char b) {
  116:   if (uart_port & USE_UART0) {
  117:     while ((IFG1 & UTXIFG0) == 0);
  118:     TXBUF0 = b;
  119:   }
  120:   else if (uart_port & USE_UART1){
  121:     while ((IFG2 & UTXIFG1) == 0);
  122:     TXBUF1 = b;
  123:   }
  124:   else return;
  125: }
```

```
126:
127: // Comment in H file
128: char uart_get_byte(void) {
129:   uart_status = TRUE;
130:   _BIS_SR(CPUOFF);
131:   uart_status = FALSE;
132:   return uart_rx_buffer;
133: }
134:
135: char uart_get_byte_timeout(void) {
136:   uart_status = TRUE;
137:   newbyte = FALSE;
138:   timer_wait(9483);
139:   uart_status = FALSE;
140:   if (newbyte == TRUE) return uart_rx_buffer;
141:   else return 0x11;
142: }
143:
144:
145: // Comment in H file
146: char uart_spi_byte(char b, int trans) {
147:   char data = 0x00;
148:   //only uart1 at the moment
149:   if (uart_port & USE_UART1){
150:     if(trans&SPI_START_TRANS) {
151:       transaction_sr = _BIS_SR(0x00);
152:       _DINT();
153:       P5OUT &= ~0x01;
154:     }
155:     uart_put_byte(b);
156:     while ((IFG2 & URXIFG1) == 0);
157:     data = RXBUF1;
158:
159:     //timer_wait(2);
160:
161:     if(trans & SPI_END_TRANS) {
162:       P5OUT |= 0x01;
163:       if(transaction_sr&GIE) _EINT();
164:     }
165:   }
166:
167:   return data;
168: }
169:
170:
171:
172: /**
173:  * Receive interrupts for UART. store value and wake CPU if it is waiting for a value,
else ignore
174:  */
175: void uart0_rx(void) __interrupt[UART0RX_VECTOR] {
176:   if(uart_port & USE_UART0 && uart_status == TRUE) {
177:     newbyte = TRUE;
178:     uart_rx_buffer = RXBUF0;
179:     _BIC_SR_IRQ(CPUOFF);
180:   }
181: }
182:
183: void uart1_rx(void) __interrupt[UART1RX_VECTOR] {
184:   if(uart_port & USE_UART1 && uart_status == TRUE) {
185:     uart_rx_buffer = RXBUF1;
186:     _BIC_SR_IRQ(CPUOFF);
187:   }
```

```
188: }
```

```
    1: #include "wireless.h"
    2: #include "cc2500.h"
    3: #include "common.h"
    4: #include "packet.h"
    5: #include "timer.h"
    6: #include "game.h"
    7: #include "keyboard.h"
    8: #include "timer.h"
    9:
   10: #include <stdio.h>
   11: /*IOCFG0 (0x06) Asserts when sync word has been sent / received, and de-asserts at the
end of the packet. In RX, the pin will de-assert
   12: when the optional address check fails or the RX FIFO overflows. In TX the pin will de-a
ssert if the TX FIFO underflows.*/
   13:
   14: #define WIRELESS_BUF_CAP 10
   15: #define WIRELESS_HEARTBEAT_MAX  20
   16:
   17: char sleep_count_hb = 0;
   18:
   19: packet heartbeat;
   20:
   21: int wireless_insert_check(packet* p);
   22:
   23: char readptr,writeptr,packetcnt;
   24: packet packetbuffer[WIRELESS_BUF_CAP];
   25:
   26: char heartbeat_from_client, heartbeat_to_client;
   27:
   28: int heartbeat_last_second = 0;
   29: int sentLastTime = FALSE;
   30: /**
   31:  * Set up variables, interrupt pins, and timer
   32:  */
   33: void wireless_init(void){
   34:   readptr = 0;
   35:   writeptr = 0;
   36:   packetcnt = 0;
   37:
   38:   heartbeat_from_client = 0;
   39:   heartbeat_to_client = 0;
   40:
   41:   P1DIR &= ~0x02;       // Port 1.1 Input, rest output
   42:   P1SEL &= ~0x02;       // From GPIO pin (when its zero its using the gpio)
   43:
   44:   // Init interrupts on Port 1
   45:   P1IFG = 0x0;
   46:   P1IES |= 0x02;
   47:   P1IE |= 0x02; //turn the interrupts on port 1.1! (I hope!)
   48:
   49:   // Start heartbeat timer
   50:   TBCTL = TBSSEL_1+TBCLR + MC1;
   51:   TBCCTL0 |= CCIE;
   52:   TBCCR0 = 32767;
   53:   TBCTL |= 0x10;
   54:
   55:   heartbeat_last_second = timer_get_seconds();
   56:
   57:   wireless_heartbeat_disable();
   58: }
   59:
   60: int wireless_packet_available(void) {
   61:   if(cc2500_state() == CC2500STATE_RXOVERFLOW) {
```

```
   62:       cc2500_flush_rx();
   63:       cc2500_strobe(CC2500REG_SRX);
   64:       while(!cc2500_state_rx()) continue;
   65:   }
   66:
   67:   return (packetcnt != 0);
   68: }
   69:
   70: void wireless_receive_status(char stat) {
   71:   if(stat) {
   72:     P1IE |= 0x02; //disable timerB
   73:   }
   74:   else {
   75:     P1IE &= ~0x02;
   76:   }
   77: }
   78:
   79: void wireless_packet_interrupt(void) __interrupt[PORT1_VECTOR] {
   80:   char type;
   81:   type = uart_switch(0);  //save previous uart mode
   82:
   83:   P1IFG = 0x0;                        //clearing the interrupt flag
   84:   //while(P1IN & 0x02) continue;      //wait until P1.1 goes low, meaning the packet ha
s been received or error ?
   85:
   86: //  _EINT();
   87:
   88: //  io_printf(IO_HYPERTERM, "Received\r\n");
   89:
   90: //  cc2500_read_packet(packetbuffer+writeptr); //putting the packet at the current writ
eptr
   91:   while(cc2500_available_packet() && cc2500_receive_packet(&packetbuffer[writeptr]) &&
packetcnt < WIRELESS_BUF_CAP) {
   92:
   93:     //io_printf(IO_HYPERTERM, "OMG\r\n");
   94:     /*If the packet received is NOT a heartbeat, keep it in the packetbuffer and increm
ent packetcnt*/
   95:     if (wireless_insert_check(&packetbuffer[writeptr])) {
   96:       //readRx(&packetbuffer[writeptr]);
   97:       writeptr = (writeptr + 1) % WIRELESS_BUF_CAP;
   98:       packetcnt++;
   99:     }
  100:   }
  101: //  io_printf(IO_HYPERTERM, "Receive done\r\n");
  102:   io_putchar(IO_HYPERTERM, '\0');
  103:
  104: //  io_printf(IO_HYPERTERM, "Packet Queue now : %i\r\n", packetcnt);
  105: //  io_printf(IO_HYPERTERM, "State now : %i\r\n", cc2500_state());
  106:
  107:   uart_switch(type);  // restore uart mode
  108:
  109:   if(timer_wake_needed()) _BIC_SR_IRQ(CPUOFF);
  110: }
  111:
  112: int wireless_packet_receive(packet* p) {
  113:   while(packetcnt == 0) continue;
  114:
  115:   memcpy(p, &packetbuffer[readptr], sizeof(packet));
  116:   readptr = (readptr + 1) % WIRELESS_BUF_CAP;
  117:   packetcnt--;
  118: }
  119:
  120: int wireless_packet_transmit(packet* p) {
```

```
121:   return cc2500_transmit_packet(p);
122: }
123:
124:
125: void wireless_heartbeat_enable(void) {
126:   heartbeat_to_client = 0;
127:   heartbeat_from_client = 0;
128:   _DINT(); // on the off chance that timer fires while enabling
129:   packet_create_z(&heartbeat, 0);
130:   _EINT();
131: }
132: void wireless_heartbeat_disable(void) {
133:   heartbeat.len = 0;
134: }
135:
136: void wireless_heartbeat_timer (void) __interrupt[TIMERB0_VECTOR]{
137:   int type;
138:
139: //  if(!sentLastTime){
140: //    _EINT();  //enable interrupts so timer_wait may be used
141:     if(heartbeat.len == 0) return;
142:
143:     type = uart_switch(0);  //save previous uart mode
144:
145:     hexdisplay_set(heartbeat_to_client);
146:
147:     heartbeat.body.z.num_beats = heartbeat_to_client;
148:     heartbeat_to_client++;
149:
150:
151:     wireless_packet_transmit(&heartbeat);
152:
153:     uart_switch(type);  // restore uart mode
154:
155:     sentLastTime = TRUE;
156:
157:     TBCCR0 = 32767;
158:
159:     if(_BIS_SR(0x00)&CPUOFF && timer_sleeping) timer_sleeping++;
160:
161:     if(timer_sleeping > 4) {
162:       _BIC_SR_IRQ(CPUOFF);
163:     }
164:
165: //  }
166: //  else{//if we sent it last time
167: //      TBCCR0 = timer_get_ticks();
168: //      sentLastTime = FALSE;
169: //  }
170:   if(timer_wake_needed()) _BIC_SR_IRQ(CPUOFF);
171: }
172:
173: int wireless_heartbeat_status(void) {
174:   return (heartbeat_to_client < WIRELESS_HEARTBEAT_MAX && heartbeat_from_client < WIREL
ESS_HEARTBEAT_MAX);
175: }
176:
177: int wireless_heartbeat_time(void) {
178:   return heartbeat_last_second;
179: }
180:
181: /*This function returns TRUE if the packet is not a HB, FALSE if it is*/
182: int wireless_insert_check(packet* p) {
```

```
183:   if(p->type == PACKET_HEARTBEAT) {
184: //    io_printf(IO_HYPERTERM, "HB\r\n");
185:
186:     heartbeat_to_client = 0;
187:     heartbeat_from_client = p->body.z.num_beats;
188:
189:     heartbeat_last_second = timer_get_seconds();
190:
191:     return FALSE;
192:   }
193:
194:   return TRUE;
195: }
```

```
 1:
 2: #include <string.h>
 3: #include "uart.h"
 4: #include "timer.h"
 5: #include "io.h"
 6: #include "packet.h"
 7:
 8: #define CC2500STATE_SLEEP 0
 9: #define CC2500STATE_IDLE  1
10:
11: #define CC2500STATE_RX0   13
12: #define CC2500STATE_RX1   14
13: #define CC2500STATE_RX2   15
14: #define CC2500STATE_RXOVERFLOW   17
15:
16: #define CC2500STATE_TX0   19
17: #define CC2500STATE_TX1   20
18: #define CC2500STATE_TXUNDERFLOW   22
19:
20:
21:
22: // Register address defines to improve code readability
23: #define CC2500REG_IOCFG2 0x00
24: #define CC2500REG_IOCFG1 0x01
25: #define CC2500REG_IOCFG0 0x02
26: #define CC2500REG_FIFOTHR 0x03
27: #define CC2500REG_SYNC1 0x04
28: #define CC2500REG_SYNC0 0x05
29: #define CC2500REG_PKTLEN 0x06
30: #define CC2500REG_PKTCTRL1 0x07
31: #define CC2500REG_PKTCTRL0 0x08
32: #define CC2500REG_ADDR 0x09
33: #define CC2500REG_CHANNR 0x0A
34: #define CC2500REG_FSCTRL1 0x0B
35: #define CC2500REG_FSCTRL0 0x0C
36: #define CC2500REG_FREQ2 0x0D
37: #define CC2500REG_FREQ1 0x0E
38: #define CC2500REG_FREQ0 0x0F
39: #define CC2500REG_MDMCFG4 0x10
40: #define CC2500REG_MDMCFG3 0x11
41: #define CC2500REG_MDMCFG2 0x12
42: #define CC2500REG_MDMCFG1 0x13
43: #define CC2500REG_MDMCFG0 0x14
44: #define CC2500REG_DEVIATN 0x15
45: #define CC2500REG_MCSM2 0x16
46: #define CC2500REG_MCSM1 0x17
47: #define CC2500REG_MCSM0 0x18
48: #define CC2500REG_FOCCFG 0x19
49: #define CC2500REG_BSCFG 0x1A
50: #define CC2500REG_AGCCTRL2 0x1B
51: #define CC2500REG_AGCCTRL1 0x1C
52: #define CC2500REG_AGCCTRL0 0x1D
53: #define CC2500REG_WOREVT1 0x1E
54: #define CC2500REG_WOREVT0 0x1F
55: #define CC2500REG_WORCTRL 0x20
56: #define CC2500REG_FREND1 0x21
57: #define CC2500REG_FREND0 0x22
58: #define CC2500REG_FSCAL3 0x23
59: #define CC2500REG_FSCAL2 0x24
60: #define CC2500REG_FSCAL1 0x25
61: #define CC2500REG_FSCAL0 0x26
62: #define CC2500REG_RCCTRL1 0x27
63: #define CC2500REG_RCCTRL0 0x28
```

```
 64: #define CC2500REG_FSTEST 0x29
 65: #define CC2500REG_PTEST 0x2A
 66: #define CC2500REG_AGCTEST 0x2B
 67: #define CC2500REG_TEST2 0x2C
 68: #define CC2500REG_TEST1 0x2D
 69: #define CC2500REG_TEST0 0x2E
 70:
 71: #define CC2500REG_SRES 0x30
 72: #define CC2500REG_SFSTXON 0x31
 73: #define CC2500REG_SXOFF 0x32
 74: #define CC2500REG_SCAL 0x33
 75: #define CC2500REG_SRX 0x34
 76: #define CC2500REG_STX 0x35
 77: #define CC2500REG_SIDLE 0x36
 78: #define CC2500REG_SWOR 0x38
 79: #define CC2500REG_SPWD 0x39
 80: #define CC2500REG_SFRX 0x3A
 81: #define CC2500REG_SFTX 0x3B
 82:
 83:
 84: #define CC2500REG_PARTNUM (0xC0|0x30)
 85: #define CC2500REG_VERSION (0xC0|0x31)
 86: #define CC2500REG_FREQEST (0xC0|0x32)
 87: #define CC2500REG_LQI     (0xC0|0x33)
 88: #define CC2500REG_RSSI    (0xC0|0x34)
 89: #define CC2500REG_MARCSTATE   (0xC0|0x35)
 90: #define CC2500REG_WORTIME1    (0xC0|0x36)
 91: #define CC2500REG_WORTIME0    (0xC0|0x37)
 92: #define CC2500REG_PKTSTATUS   (0xC0|0x38)
 93: #define CC2500REG_VCO_VC_DAC  (0xC0|0x39)
 94: #define CC2500REG_TXBYTES     (0xC0|0x3A)
 95: #define CC2500REG_RXBYTES     (0xC0|0x3B)
 96: #define CC2500REG_RCCTRL1_STATUS  (0xC0|0x3C)
 97: #define CC2500REG_RCCTRL0_STATUS  (0xC0|0x3D)
 98:
 99:
100: #define CC2500REG_DATABUF 0x3F
101:
102: #define CC2500_UART (USE_UART1|USE_SPI)
103:
104: #ifndef __INCLUDE_CC2500_H
105: #define __INCLUDE_CC2500_H
106:
107: #define CC2500_WRITE 0x00
108: #define CC2500_READ 0x80
109: #define CC2500_BURST 0x40
110:
111: typedef struct {
112:   char address;
113:   char* name;
114: } cc2500_reg;
115:
116: /**
117:  * Please see C file for comments on functions
118:  */
119:
120: void cc2500_init(void);
121: char cc2500_status(void);
122: char cc2500_read_reg(char i);
123: void cc2500_write_reg(char i, char val);
124: void cc2500_reset(void);
125: int cc2500_verify(void);
126:
```

```
127: void cc2500_setradioregisters(void);
128:
129:
130: char* cc2500_reg_get_name(int addr);
131: int cc2500_reg_get_address(char* name);
132:
133: int cc2500_available_packet(void);
134: int cc2500_receive_packet(packet* p);
135: int cc2500_transmit_packet(packet* p);
136:
137: int cc2500_read_packet(packet* p);
138: void cc2500_write_packet(packet* p);
139:
140:
141: void cc2500_strobe(char strobe);
142:
143:
144: int cc2500_state(void);
145: int cc2500_state_idle(void);
146: int cc2500_state_rx(void);
147: int cc2500_state_tx(void);
148:
149: int cc2500_rx_size(void);
150: int cc2500_tx_size(void);
151: void cc2500_flush_rx(void);
152: void cc2500_flush_tx(void);
153:
154: #endif
155:
```

```
 1: /**
 2:  * Register values for CC2500. Used to print text names of registers
 3:  *
 4:  * Only included in cc2500.c DO NOT include in anything else, as will result in linking
error
 5:  */
 6:
 7:
 8: #include "cc2500.h"
 9:
10:
11: cc2500_reg registers[] = {
12:   {CC2500REG_IOCFG2,"IOCFG2"},
13:   {CC2500REG_IOCFG1,"IOCFG1"},
14:   {CC2500REG_IOCFG0,"IOCFG0"},
15:   {CC2500REG_FIFOTHR,"FIFOTHR"},
16:   {CC2500REG_SYNC1,"SYNC1"},
17:   {CC2500REG_SYNC0,"SYNC0"},
18:   {CC2500REG_PKTLEN,"PKTLEN"},
19:   {CC2500REG_PKTCTRL1,"PKTCTRL1"},
20:   {CC2500REG_PKTCTRL0,"PKTCTRL0"},
21:   {CC2500REG_ADDR,"ADDR"},
22:   {CC2500REG_CHANNR,"CHANNR"},
23:   {CC2500REG_FSCTRL1,"FSCTRL1"},
24:   {CC2500REG_FSCTRL0,"FSCTRL0"},
25:   {CC2500REG_FREQ2,"FREQ2"},
26:   {CC2500REG_FREQ1,"FREQ1"},
27:   {CC2500REG_FREQ0,"FREQ0"},
28:   {CC2500REG_MDMCFG4,"MDMCFG4"},
29:   {CC2500REG_MDMCFG3,"MDMCFG3"},
30:   {CC2500REG_MDMCFG2,"MDMCFG2"},
31:   {CC2500REG_MDMCFG1,"MDMCFG1"},
32:   {CC2500REG_MDMCFG0,"MDMCFG0"},
33:   {CC2500REG_DEVIATN,"DEVIATN"},
34:   {CC2500REG_MCSM2,"MCSM2"},
35:   {CC2500REG_MCSM1,"MCSM1"},
36:   {CC2500REG_MCSM0,"MCSM0"},
37:   {CC2500REG_FOCCFG,"FOCCFG"},
38:   {CC2500REG_BSCFG,"BSCFG"},
39:   {CC2500REG_AGCCTRL2,"AGCCTRL2"},
40:   {CC2500REG_AGCCTRL1,"AGCCTRL1"},
41:   {CC2500REG_AGCCTRL0,"AGCCTRL0"},
42:   {CC2500REG_WOREVT1,"WOREVT1"},
43:   {CC2500REG_WOREVT0,"WOREVT0"},
44:   {CC2500REG_WORCTRL,"WORCTRL"},
45:   {CC2500REG_FREND1,"FREND1"},
46:   {CC2500REG_FREND0,"FREND0"},
47:   {CC2500REG_FSCAL3,"FSCAL3"},
48:   {CC2500REG_FSCAL2,"FSCAL2"},
49:   {CC2500REG_FSCAL1,"FSCAL1"},
50:   {CC2500REG_FSCAL0,"FSCAL0"},
51:   {CC2500REG_RCCTRL1,"RCCTRL1"},
52:   {CC2500REG_RCCTRL0,"RCCTRL0"},
53:   {CC2500REG_FSTEST,"FSTEST"},
54:   {CC2500REG_PTEST,"PTEST"},
55:   {CC2500REG_AGCTEST,"AGCTEST"},
56:   {CC2500REG_TEST2,"TEST2"},
57:   {CC2500REG_TEST1,"TEST1"},
58:   {CC2500REG_TEST0,"TEST0"},
59:
60:   {0xC0|0x30,"PARTNUM"},
61:   {0xC0|0x31,"VERSION"},
62:   {0xC0|0x32,"FREQEST"},
```

```
63:    {0xC0|0x33,"LQI"},
64:    {0xC0|0x34,"RSSI"},
65:    {0xC0|0x35,"MARCSTATE"},
66:    {0xC0|0x36,"WORTIME1"},
67:    {0xC0|0x37,"WORTIME0"},
68:    {0xC0|0x38,"PKTSTATUS"},
69:    {0xC0|0x39,"VCO_VC_DAC"},
70:    {0xC0|0x3A,"TXBYTES"},
71:    {0xC0|0x3B,"RXBYTES"},
72:    {0xC0|0x3C,"RCCTRL1_STATUS"},
73:    {0xC0|0x3D,"RCCTRL0_STATUS"},
74: };
```

```
 1:
 2: #ifndef __INCLUDE_COMMON_H
 3: #define __INCLUDE_COMMON_H
 4:
 5: #include <msp430x14x.h>
 6: #include <stdlib.h>
 7: #include <cross_studio_io.h>
 8:
 9: #define TRUE        1
10: #define FALSE       0
11:
12: #endif
```

```
 1:
 2: #include  <msp430x14x.h>
 3:
 4:
 5: #ifndef __MEMORY_H
 6: #define __MEMORY_H
 7:
 8:
 9:
10: /* Set Flash Timing Generator
11:   - Set to MCLK/2 for Flash Timing Generator
12: */
13: void setFlashTimingGen(void);
14: void  writeToFlashMemory(char value, int offset);
15: char  readFromFlashMemory(int address);
16:
17:
18:
19:
20: #endif
```

```
 1: #ifndef __INCLUDE_GAME_H
 2: #define __INCLUDE_GAME_H
 3:
 4: #include "packet.h"
 5:
 6: // Common states
 7: #define PRINT_INIT 5
 8: #define HOST_OR_JOIN 8
 9:
10: // Host states
11: #define SEND_HOST_BEACON 10
12: #define DO_YOU_ACCEPT 11
13: #define SET_UP_HEARTBEATS 12
14:
15: // Client states
16: #define LISTEN_FOR_BEACON 20
17: #define ACCEPT_BEACON 21
18: #define WAIT_HOST_HB 22
19:
20: // Game states
21: #define INIT_BOARD 31
22: #define SET_NUMBER 32
23: #define SEND_DATA 33
24: #define GRAB_INPUT 34
25: #define GAME_OVER 41
26:
27:
28:
29: void playGame(void);
30: void sendHeartbeat(void);
31: void interpret_response(packet *response, int *current_state); //formerly readRx
32:
33:
34:
35: #endif
```

```
 1: #ifndef __INCLUDE_HYPERTERM_H
 2: #define __INCLUDE_HYPERTERM_H
 3:
 4: void hyperterm_init(void);
 5:
 6: void hyperterm_set_echo(int stat);
 7: void hyperterm_clear(void);
 8:
 9: void hyperterm_putchar(char c);
10: char hyperterm_getchar(void);
11: char hyperterm_timeoutgetchar(void);
12:
13: #endif
```

```
 1:
 2: #define IO_KEYBOARD 0x02
 3: #define IO_HYPERTERM 0x04
 4: #define IO_LCD 0x08
 5:
 6: #ifndef __INCLUDE_IO_H
 7: #define __INCLUDE_IO_H
 8:
 9: void io_init(void);
10:
11: void io_putchar(int type, char c);
12: int io_getchar(int type);
13: int io_timeoutgetchar(int type);
14:
15: void io_printf(int type, char* format, ...);
16: void io_clear(int type);
17:
18: int io_gethex(int type);
19:
20: #endif
```

```
    1:
    2:
    3: #ifndef __INCLUDED_KEYBOARD_H
    4: #define __INCLUDED_KEYBOARD_H
    5:
    6: void hexdisplay_set(char b);
    7:
    8: void keyboard_init(void);
    9: char keyboard_getchar(void);
   10: char keyboard_timeoutgetchar(void);
   11:
   12: int keyboard_make_code(char* scancode);
   13: int keyboard_scancode_release(char* code);
   14: void keyboard_wait_scancode(char* code);
   15: void keyboard_receive_scancode(char* code);
   16:
   17: int keyboard_make_code_to_ascii(int make_code);
   18:
   19: #endif
```

```
 1: /**
 2:  * LCD header file with address constants to call in the lcd_goto() function
 3:  *
 4:  */
 5:
 6: #ifndef __INCLUDE_LCD_H
 7: #define __INCLUDE_LCD_H
 8:
 9: #define LCD_LINE1 0
10: #define LCD_LINE2 24
11:
12: void lcd_init(void);
13: void lcd_putchar(char c);
14: char lcd_read_status(void);
15: void printString(char* string);
16: void lcd_goto(char p);
17: void lcd_pulse_e(void);
18: void send_nibble(unsigned char data);
19: void send_data(unsigned char data);
20: void lcd_clearscreen(void);
21: char lcd_printf(char* format, ...);
22: int lcd_checkAlphabet(char key);
23: void lcd_changeline(void);
24: void lcd_clearline(int line);
25: #endif
26:
```

```
 1:
 2: #ifndef __INCLUDE_PACKET_H
 3: #define __INCLUDE_PACKET_H
 4:
 5: #define PACKET_H 'H'
 6: #define PACKET_R 'R'
 7: #define PACKET_Z 'Z'
 8: #define PACKET_G 'G'
 9: #define PACKET_Y 'Y'
10: #define PACKET_S 'S'
11:
12: #define PACKET_HOST_BEACON       PACKET_H
13: #define PACKET_JOIN_REQUEST      PACKET_R
14: #define PACKET_HEARTBEAT         PACKET_Z
15: #define PACKET_GUESS             PACKET_G
16: #define PACKET_GUESS_RESPONSE    PACKET_Y
17: #define PACKET_SERVER_MESSAGE    PACKET_S
18:
19:
20: typedef struct {
21:   char len;
22:   char dest;
23:   char src;
24:
25:   char type;
26:
27:   //a union is like a struct, except each member uses the same memory space
28:   // so you should only set one of the variables at a time
29:   union {
30:
31:     struct {     // 20 bytes, Host beacon
32:       char name[20];
33:     } h;
34:
35:     struct {     // 20 bytes, Join Request
36:       char name[20];
37:     } r;
38:
39:     struct {     // 1 byte, Heartbeat
40:       char num_beats;
41:     } z;
42:
43:     struct {     // 6 bytes, Guess
44:       int guess_count;
45:       char guess[4];
46:     } g;
47:
48:     struct {     // 2 bytes, Guess reponse
49:       char num_bulls;
50:       char num_cows;
51:     } y;
52:
53:     struct {     // 31 bytes, Server Message
54:       char t;
55:       char message[30];
56:     } s;
57:
58:   } body;
59:
60:   //these are not alligned as in FIFO and will be set manually
61:   char rssi;
62:   char lqi;
63:
```

```
64: } packet;
65:
66:
67: void packet_print(int out_type, packet *p);
68:
69: void packet_create_h(packet* p, char* name);
70: void packet_create_r(packet* p, char* name);
71: void packet_create_z(packet* p, char num_beats);
72: void packet_create_g(packet* p, int guess_count, char* guess);
73: void packet_create_y(packet* p, char bulls, char cows);
74: void packet_create_s(packet* p, char type, char* mess);
75:
76:
77: void packet_source(char src);
78: void packet_destination(char dest);
79:
80: #endif
81:
```

```
 1: /**
 2:  * Timer H file for wait and get_ticks function
 3:  *
 4:  * Expects:
 5:  * 1. TimerA to be running at 32768Hz
 6:  * 2. TimerA counting up and interrupt A1 is not in use
 7:  * 3. TimerA will be run continuously
 8:  */
 9:
10:
11: #ifndef __INCLUDE_TIMER_H
12: #define __INCLUDE_TIMER_H
13:
14: extern char timer_sleeping;
15:
16: void timer_init(void);
17: void timer_wait(int ticks);
18: void timer_wait_seconds(int seconds);
19: char timer_waiting(void);
20:
21: char timer_wake_needed(void);
22:
23: void timer_reset(void);
24: int timer_get_ticks(void);
25: int timer_get_seconds(void);
26:
27:
28: void initHeartbeatTimer(void);
29: #endif
```

```
 1: /**
 2:  * UART H file for send bytes
 3:  *
 4:  * Expects:
 5:  * 1. SMCLK to be running at 8MHz
 6:  * 2. Port 1 and 3 attached to the keypad and P1 interrupt not in use
 7:  * 3. If SPI, Port 3 and 5 not in use elsewhere and ready for use
 8:  * 4. TimerA counting up and interrupt A0 is not in use
 9:  */
10:
11: #ifndef __INCLUDE_UART_H
12: #define __INCLUDE_UART_H
13:
14: #define USE_UART0 0x1
15: #define USE_UART1 0x2
16: #define USE_SPI 0x4
17:
18:
19: #define SPI_CONT_TRANS 0x0
20: #define SPI_START_TRANS 0x1
21: #define SPI_END_TRANS 0x2
22: #define SPI_SINGLE_TRANS (SPI_START_TRANS|SPI_END_TRANS)
23:
24:
25: /**
26:  * Select which UART port the next uart method will effect and return previous val
27:  */
28: void uart_init(void);
29:
30: /**
31:  * Initialize the UART control registers to a known value and configure for 57600Kbps,
no parity, 8-bit, SPI optional
32:  */
33: int uart_switch(int type);
34:
35: /**
36:  * Put a byte on the UART data lines
37:  */
38: void uart_put_byte(char b);
39:
40: /**
41:  * Retrieve a value from the uart data lines
42:  */
43: char uart_get_byte(void);
44: char uart_get_byte_timeout(void);
45:
46: /**
47:  * Send and receive a value via SPI, and optionally pull STE low or high or neither
48:  */
49: char uart_spi_byte(char b, int trans);
50:
51: #endif
```

```
 1: #include "packet.h"
 2:
 3: #ifndef __INCLUDE_WIRELESS_H
 4: #define __INCLUDE_WIRELESS_H
 5:
 6:
 7: void wireless_init(void);
 8:
 9: int wireless_packet_available(void);
10: int wireless_packet_receive(packet* p);
11: int wireless_packet_transmit(packet* p);
12:
13: void wireless_receive_status(char stat);
14:
15: int wireless_heartbeat_status(void);
16: void wireless_heartbeat_enable(void);
17: void wireless_heartbeat_disable(void);
18: int wireless_heartbeat_time(void);
19:
20: #endif
```

```
 1: -- CPLD pin passthrough
 2: -- Copyright (C) 2009 M. Perreault
 3: -- Version 1.0
 4: -- Author: M. Perreault
 5: -- Date: October 8th, 2009
 6:
 7:
 8: library ieee; -- allows use of the std_logic_vector type
 9: use ieee.std_logic_1164.all;
10:
11: entity CPLD430 is
12: Port (
13:   -- PORT(1,2,4,5) to MSP, can be changed to outputs
14:   -- PORT5(7) should map to SPI_clk
15:   PORT1_0: in std_logic;
16:   PORT1_1: in std_logic;
17:   PORT1_2: in std_logic;
18:   PORT1_3: in std_logic;
19:   PORT1_4: in std_logic;
20:   PORT1_5: in std_logic;
21:   PORT1_6: in std_logic;
22:   PORT1_7: in std_logic;
23:
24:   PORT2_0: out std_logic;
25:   PORT2_1: in std_logic;
26:   PORT2_2: in std_logic;
27:   PORT2_3: in std_logic;
28:   PORT2_4: in std_logic;
29:   PORT2_5: in std_logic;
30:   PORT2_6: in std_logic;
31:   PORT2_7: in std_logic;
32:
33:   PORT4_0: in std_logic;
34:   PORT4_1: in std_logic;
35:   PORT4_2: in std_logic;
36:   PORT4_3: in std_logic;
37:   PORT4_4: in std_logic;
38:   PORT4_5: in std_logic;
39:   PORT4_6: in std_logic;
40:   PORT4_7: in std_logic;
41:
42:   PORT5_0: in std_logic;
43:   PORT5_1: in std_logic;
44:   PORT5_2: out std_logic;
45:   PORT5_3: in std_logic;
46:   PORT5_4: in std_logic;
47:   PORT5_5: in std_logic;
48:   PORT5_6: in std_logic;
49:   --PORT5_7: in std_logic;
50:   --OUTA(7) should map to clk_en
51:   --OUTA(7 downto 0) should map to expansion port
52:   --(pins 3, 5, 7, 9, 10, 8, 6, 4)
53:   OUTA_0 : out std_logic;
54:   OUTA_1 : out std_logic;
55:   OUTA_2 : in std_logic;
56:   OUTA_3 : out std_logic;
57:   OUTA_4 : out std_logic;
58:   OUTA_5 : out std_logic;
59:   OUTA_6 : out std_logic;
60:   --OUTB(0) maps to segA (drive LO to light; port pin 17)
61:   --OUTB(1) maps to segB (port pin 19)
62:   --OUTB(2) maps to segC (port pin 21)
63:   --OUTB(3) maps to segD (port pin 23)
```

```
 64:    --OUTB(4) maps to segE (port pin 25)
 65:    --OUTB(5) maps to segF (port pin 27)
 66:    --OUTB(6) maps to segG (port pin 29)
 67:    OUTB: out std_logic_vector(6 downto 0);
 68:    --OUTB_0 : out std_logic; --segA
 69:    --OUTB_1 : out std_logic; --segB
 70:    --OUTB_2 : out std_logic; --segC
 71:    --OUTB_3 : out std_logic; --segD
 72:    --OUTB_4 : out std_logic; --segE
 73:    --OUTB_5 : out std_logic; --segF
 74:    --OUTB_6 : out std_logic; --segG
 75:    --OUTC(4) maps to segDP (port pin 35)
 76:    --OUTC(5) maps to Common Anode 1 (HI when in use; port pin 31)
 77:    --OUTC(6) maps to Common Anode 2 (HI when in use; port pin 33)
 78:    --OUTC(3 downto 0) to expansion port (pins 18, 16, 14, 12)
 79:    OUTC_0: out std_logic;
 80:    OUTC_1: out std_logic;
 81:    OUTC_2: out std_logic;
 82:    OUTC_3: out std_logic;
 83:    OUTC_4: out std_logic; -- segDP
 84:    OUTC_5: out std_logic; -- CA1
 85:    OUTC_6: out std_logic; -- CA2
 86:    --OUTD(6 downto 0) to expansion port
 87:    --(pins 36, 34, 32, 30, 28, 26, 24)
 88:    OUTD_0: out std_logic;
 89:    OUTD_1: out std_logic;
 90:    OUTD_2: in std_logic;
 91:    OUTD_3: out std_logic;
 92:    OUTD_4: out std_logic;
 93:    OUTD_5: in std_logic;
 94:    OUTD_6: in std_logic;
 95:
 96:    CLK_en : out std_logic;
 97:    --oscillator enable, used with aux clock
 98:
 99:    CLK : in std_logic;
100:    -- used as global clock
101:
102:    --SPI_clk : in std_logic; --not sure this is mapped right
103:    -- aux clock
104:
105:    GLB_RSTn : in std_logic;
106:    -- internal register reset (active low, pulled HI)
107:
108:    OE : in std_logic
109:    -- pulled up normally, bypass to expansion port pin 22
110:  );
111:  end CPLD430;
112:
113:  architecture arch1 of CPLD430 is
114:
115:    component PS2Interface is
116:    port(
117:            ps2_clk       : in std_logic;
118:            ps2_data      : in std_logic;
119:            scancode      : out std_logic_vector(23 downto 0);
120:            key_event     : out std_logic;
121:            state         : out std_logic_vector(1 downto 0)
122:    );
123:    end component;
124:
125:    component segdecoder is
126:    port(
```

```
127:        segs_out  : out std_logic_vector(6 downto 0);
128:        char_in   : in  std_logic_vector(3 downto 0)
129:          );
130:    end component;
131:
132:    signal ste: std_logic;
133:    signal keycode: std_logic_vector(23 downto 0);
134:    signal spi_transmit: std_logic_vector(23 downto 0) := "000000000000000000000000";
135:    signal spi_receive: std_logic_vector(7 downto 0);
136:
137:    signal key_event: std_logic := '0';
138:
139:    signal ps2_state: std_logic_vector(1 downto 0);
140:
141:    signal clk_buffer : std_logic_vector(7 downto 0);
142:    signal ps2_clk : std_logic;
143:
144:    signal decoder_values: std_logic_vector (7 downto 0) := "11001010";
145:    signal segcode: std_logic_vector (3 downto 0);
146:    signal ca_state: std_logic;
147:    signal clock_count : integer range 0 to 255;
148:
149:  begin
150:    CLK_en <= '1';
151:
152:    OUTA_4 <= ste;                  -- ste
153:    OUTA_5 <= PORT5_1;      --simo
154:    OUTA_6 <= spi_transmit(23);    --somi
155:    OUTC_0 <= PORT5_3;      --spi_clk
156:
157:    OUTA_1 <= PORT5_1; -- cc2500 SIMO
158:    OUTA_3 <= PORT5_3; -- cc2500 spi_clk
159:
160:    PORT2_0 <= key_event;
161:
162:
163:    with clk_buffer select
164:      ps2_clk <=  '1' WHEN "11111111",
165:                                '0' when "00000000",
166:                                ps2_clk when others;
167:    clk_buf: process(OUTD_2) -- 40MHz clk
168:    begin
169:      if(rising_edge(OUTD_2)) then
170:
171:        clock_count <= clock_count +1;
172:        if(clock_count = 255) then
173:                          ca_state <= not ca_state;
174:                          end if;
175:
176:            clk_buffer <= clk_buffer(6 downto 0) & OUTD_5;
177:          end if;
178:    end process;
179:
180:
181:    ps2: PS2Interface
182:    port map(
183:      ps2_clk,     --clk
184:      OUTD_6,      --data
185:      keycode,     --code
186:      key_event,   --key sensed
187:          ps2_state --state
188:    );
189:
```

```
190:   --7-segment decoder code
191:   with ca_state select
192:         segcode <= decoder_values(7 downto 4) when '1',
193:                    decoder_values(3 downto 0) when '0',
194:                    "0000" when others;
195:
196:   OUTC_4 <= '1';
197:   OUTC_5 <= ca_state; --CA1, right digit ?
198:   OUTC_6 <= not ca_state; --CA2, left digit?
199:
200:   segmentdecoder: segdecoder
201:   port map(OUTB,segcode);
202:   --
203:
204:   -- 0 on PORT5_6 selects H1 cc2500, 1 selects internal SPI
205:   WITH PORT5_6 SELECT
206:     PORT5_2 <=  OUTA_2                          WHEN '0', -- cc2500 somi
207:                                 spi_transmit(23)        WHEN others; -- CPLD somi
208:
209:   WITH PORT5_6 SELECT
210:     OUTA_0 <=   PORT5_0 WHEN '0', -- cc2500 ste
211:                                 '1'     WHEN others;
212:   WITH PORT5_6 SELECT
213:     ste <=      '1'     WHEN '0',
214:                         PORT5_0 WHEN others;    -- CPLD ste
215:
216:   OUTD_0 <= CLK;
217:
218:   OUTD_3 <= OUTD_5;
219:   OUTD_4 <= OUTD_6;
220:
221:   spi_send: process(PORT5_3, ste, keycode)
222:   begin
223:         if(ste = '1') then
224:                 spi_transmit <= keycode;
225:                 decoder_values <= spi_receive;
226:         else
227:                 if(PORT5_3'event and PORT5_3 = '1') then
228:                         spi_receive(7 downto 0) <= spi_receive(6 downto 0) & PORT5_1; -
- SIMO
229:                 elsif(PORT5_3'event and PORT5_3 = '0') then
230:                         spi_transmit(23 downto 1) <= spi_transmit(22 downto 0);
231:                 end if;
232:         end if;
233:   end process;
234:
235: end arch1;
```

```
    1: library ieee;
    2: use ieee.std_logic_1164.all;
    3:
    4: entity PS2Interface is
    5: port(
    6:         ps2_clk : in std_logic;
    7:         ps2_data        : in std_logic;
    8:         scancode        : out std_logic_vector(23 downto 0);
    9:         key_event       : out std_logic;
   10:         state           : out std_logic_vector(1 downto 0)
   11: );
   12: end PS2Interface;
   13:
   14: architecture arch of PS2Interface is
   15:         type key_states is (START,RECEIVE_BIT,PARITY,STOP);
   16:
   17:         signal data_bytes       : std_logic_vector(23 downto 0) := "0000000000000000000
00000";
   18:         signal key_fsm          : key_states := START;
   19:         signal parity_bit       : std_logic;
   20:         signal index : integer range 0 to 7;
   21: begin
   22:         WITH key_fsm SELECT
   23:                 state <=        "00" when START,
   24:                                 "01" when RECEIVE_BIT,
   25:                                 "10" when PARITY,
   26:                                 "11" when others;
   27:
   28:
   29:         monitor: process(ps2_clk)
   30:         begin
   31:                 if(falling_edge(ps2_clk)) then
   32:
   33:
   34:                         case key_fsm is
   35:                         when START =>
   36:                                 if(ps2_data = '0') then
   37:                                         parity_bit <= '1';
   38:                                         key_fsm <= RECEIVE_BIT;
   39:                                         key_event <= '0';
   40:                                         data_bytes(7 downto 0) <= (others => '0');
   41:                                         index <= 0;
   42:                                 end if;
   43:                         when RECEIVE_BIT =>
   44:                                 data_bytes(7 downto 0) <= ps2_data & data_bytes(7 downt
o 1); --it sends LSB first
   45:                                 parity_bit <= ps2_data xor parity_bit;
   46:                                 if(index = 7) then
   47:                                         key_fsm <= PARITY;
   48:                                 else
   49:                                         index <= index + 1;
   50:                                 end if;
   51:                         when PARITY =>
   52:                                 if(parity_bit = ps2_data) then
   53:                                         key_fsm <= STOP;
   54:                                 else
   55:                                         key_fsm <= START;
   56:                                         --this means the parity was wrong, but whatever
   57:                                         data_bytes <= (others => '0');
   58:                                 end if;
   59:                         when STOP =>
   60:                                 if(ps2_data = '1') then
   61:                                         key_fsm <= START;
```

```
   62:                                         if(data_bytes(7 downto 0) = "11100000" or data_
bytes(7 downto 0) = "11110000") then    --if E0 or F0, assume another byte is coming
   63:                                                 data_bytes(23 downto 8) <= data_bytes(1
5 downto 0);
   64:                                                 data_bytes(7 downto 0) <= "00000000";
   65:                                         else
   66:                                                 scancode <= data_bytes;
   67:                                                 data_bytes <= (others => '0');
   68:                                                 key_event <= '1';
   69:                                         end if;
   70:                                 else
   71:                                         data_bytes <= (others => '0');
   72:                                         --something is wrong if this is happening
   73:                                 end if;
   74:                         end case;
   75:                 end if;
   76:         end process;
   77:
   78: end arch;
```

```
 1: library ieee;
 2: use ieee.std_logic_1164.all;
 3:
 4: entity segdecoder is
 5: port(
 6:       segs_out  : out std_logic_vector(6 downto 0);
 7:       char_in   : in  std_logic_vector(3 downto 0)
 8: );
 9: end segdecoder;
10:
11: architecture a1 of segdecoder is
12:
13: begin
14:
15: with char_in select
16:
17: segs_out <= "1000000" when "0000",
18:                         "1111001" when "0001",
19:                         "0100100" when "0010",
20:                         "0110000" when "0011",
21:                         "0011001" when "0100",
22:                         "0010010" when "0101",
23:                         "0000010" when "0110",
24:                         "1111000" when "0111",
25:                         "0000000" when "1000",
26:                         "0011000" when "1001",
27:                         "0001000" when "1010",
28:                         "0000011" when "1011",
29:                         "1000110" when "1100",
30:                         "0100001" when "1101",
31:                         "0000110" when "1110",
32:                         "0001110" when "1111",
33:                         "1111111" when others;
34:
35: end a1;
36:
```