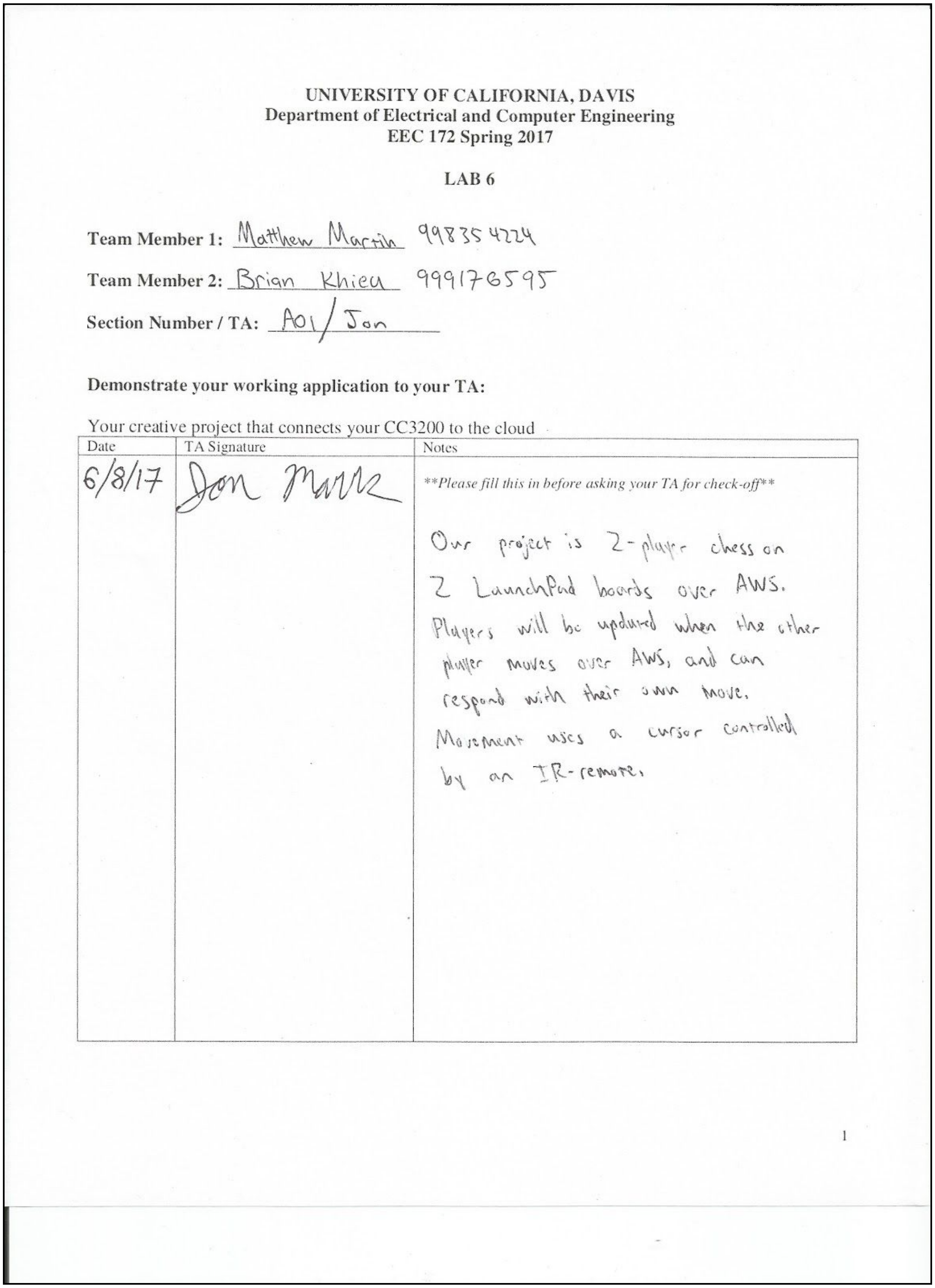
Matthew Martin, 998354224
Brian Khieu, 999176595
June 9, 2017

EEC 172 - Lab 6: Open ended IoT project - Multiplayer Chess



UNIVERSITY OF CALIFORNIA, DAVIS
Department of Electrical and Computer Engineering
EEC 172 Spring 2017

LAB 6

Team Member 1: Matthew Martin 998354224

Team Member 2: Brian Khieu 999176595

Section Number / TA: A01 / Jon

Demonstrate your working application to your TA:

Your creative project that connects your CC3200 to the cloud

Date	TA Signature	Notes
6/8/17	Jon Marr	***Please fill this in before asking your TA for check-off*** Our project is 2-player chess on 2 LaunchPad boards over AWS. Players will be updated when the other player moves over AWS, and can respond with their own move. Movement uses a cursor controlled by an IR-remote.

Introduction / Background

This lab is meant to showcase the skills we have acquired in working with embedded systems and connecting to Amazon Web Services and RESTful APIs. We were tasked with using the aforementioned technologies to produce something inventive. We chose to implement multiplayer chess using two CC3200 boards that send board states to each other using AWS.

Goals

The goal of this lab is to create a system that utilizes the CC3200 launchpads and a secure connection to Amazon Web Services in order to produce an application that demonstrates the knowledge we have gained from this course. This lab is intended to have us demonstrate our creativity and expertise with embedded systems and implementing an IoT device.

We decided to create a two-player chess application. Our goal was to be able to play chess using two boards that were connected through AWS, allowing two users to play against each other from anywhere with an internet connection. We wanted to be able to play a full game of chess with an interface that does not allow users to make illegal moves, displays legal moves to users, recognizes when the game is finished (a player is in checkmate), and allows a finished game to be restart.

We wanted to use an IR remote for user input, and an Adafruit OLED screen as a display.

Methods

1. First, we designed each chess piece for rendering on the 128x128 OLED screen. We decided to render the chess board across the entire screen, so each piece would be 16x16 pixels (the board needs to be 8x8 squares).
2. We designed each piece pixel-by-pixel, using Microsoft Paint.
3. These piece images were then encoded (by hand) in arrays for use within the program. Each piece is represented by a 6x6 boolean array (pieces do not take up the whole square to allow for a border).
4. A function was written to place a particular piece with a player’s color, from the point of view of a specific player (the black player sees the board upside-down compared to the white player).
5. Next, a global 8x8 2D character array was created to store the current local board state. The letters ‘B’, ‘K’, ‘N’, ‘P’, ‘Q’, and ‘R’ were used to represent a bishop, king, knight, pawn, queen, and rook respectively. Uppercase characters are used for the white player and lowercase for the black player. The ‘0’ character denotes an empty space.

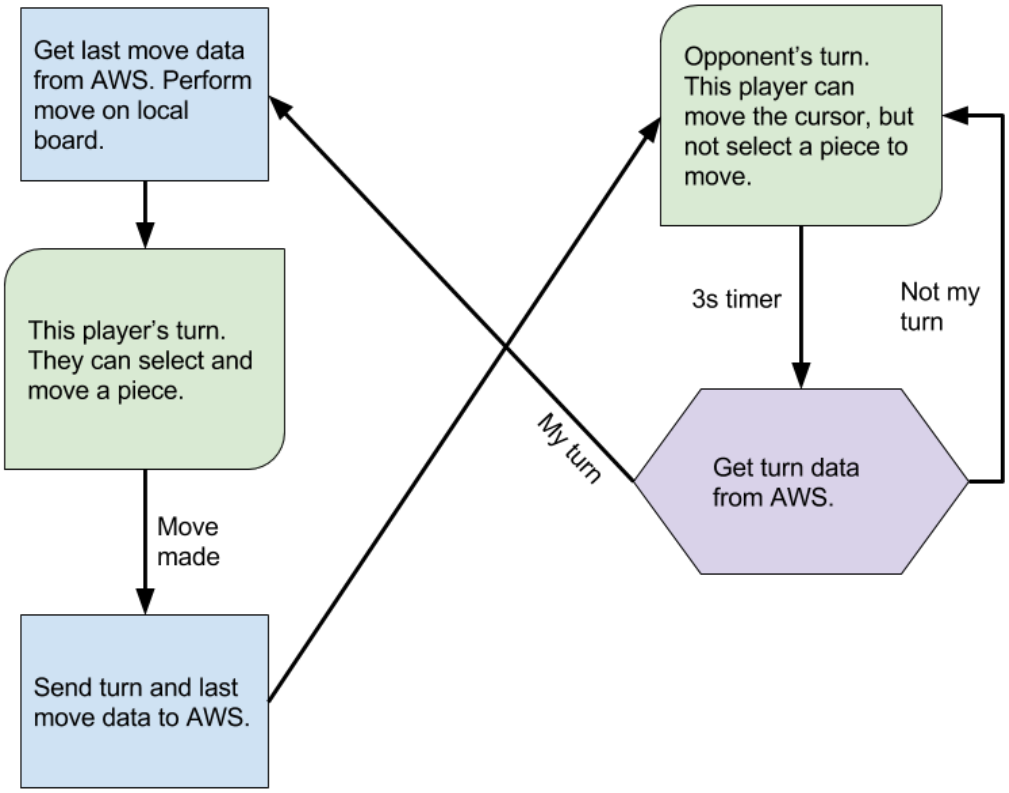
6. Functions were written to render the alternating-color background along with the current state of the board with pieces at their correct positions.
7. A cursor system was developed in order for users to select which piece they want to move. The cursor is controlled by an IR-remote, using the code we developed in Lab 3.
8. A function, validMove, was created to check if a move is legal. This function takes the player (black or white), board state, and source and destination coordinates in order to determine if a move is allowed. The movement scheme of each piece is encoded into this function. In addition, a parameter can be passed to specify whether the player ending in “check” disqualifies the move or not (both versions of the function were later used).
9. Logic was added to render the valid moves of the currently selected piece (or the piece the cursor is on, if no piece is selected) using a green selection box. The currently selected piece was highlighted in blue.
10. Next, the provided http_post function was modified to accept a “field” and “value” as parameters. When the function is called, it pushes the field-value pair into the device shadow on AWS.
11. An http_get function was added, modeled off the http_post function, that takes the “field” as a parameter and returns the value currently in this field in the device shadow. This was accomplished by querying AWS for the current state, and parsing the return data to find the correct field value.
12. The program was modified to push a “turn” and “lastmove” field to the device shadow on AWS when a move is made. The “turn” field consists of a single character representing whose turn it is (‘B’ or ‘W’), and the “lastmove” field consists of 4 characters representing the x and y coordinates of the source and destination tile of the last move that was made.
13. After a player makes a move, the program enters an infinite polling loop that performs the http_get function every few seconds with the “turn” field. Once the http_get reports that the “turn” field now corresponds to this player again (after the opponent moves and updates AWS), another http_get is performed to fetch the last move. This move is performed on the local copy of the board state, and the player is allowed to make their move.
14. At this point, the game worked fully, allowing two players to play against each other. Additional logic was added to detect when a player is in checkmate (the game is over), and allow users to restart the game when this occurs.
15. Lastly, a message system was added to notify users when it is their turn, or they try to perform an illegal action, for instance.

Discussion

In order to achieve our goal of online multiplayer chess, we started with the frameworks from Lab 3 and Lab 5. In Lab 3 introduced the functionality of the IR remote, and Lab 5 connected to the AWS IoT service. We modified the Lab 3 functionality to move a cursor around the 8x8 chess board, and modified Lab 5 to send and get data to/from the AWS servers.

Design

The program operates like a state machine, changing states based on user input and AWS get-data. Each board is flashed to behave as a different player - 1 black and 1 white. The white player starts in the “my turn” state, and the black player starts in the “opponent turn” state. A diagram of the state machine is shown below.


Get last move data from AWS. Perform move on local board.
This player’s turn. They can select and move a piece.
Move made
Send turn and last move data to AWS.
Opponent’s turn. This player can move the cursor, but not select a piece to move.
3s timer
Not my turn
Get turn data from AWS.
My turn

This design allows us to synchronize the boards’ actions, and respond to one another without introducing synchronization issues. The AWS communication was achieved by sending HTTP POST and GET commands to a single device shadow in the AWS IoT module.

Program Flow

The program starts by initializing all the components, from the OLED to the AWS connection. After this initialization is complete, it enters in infinite loop for the remainder of the program. This loop simply queries AWS for the turn information if it is not the current player's turn. The remainder of the functionality of the program is accomplished through GPIO interrupts that are triggered by the IR remote, and timers that help the IR remote detect which button was pressed. When a button on the remote is pressed, it simply calls a function to execute the command within the interrupt handler. We were advised against updating the OLED or making AWS calls within an interrupt handler; we could have implemented a queuing system where commands are simply pushed into the queue by the ISR and the commands are executed in the main program loop. However, leaving the control flow as it was for Lab 3 did not introduce any problems for us, besides a very small lag in IR control after a long-running command. This did not present a big enough issue for us to rework the control flow.

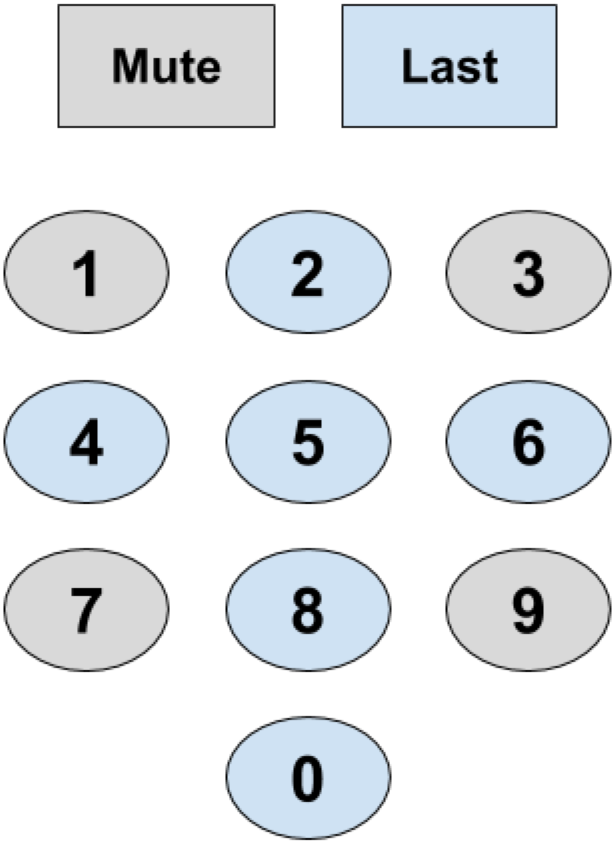
We implemented 5 primary functions to handle the game logic and AWS communication. They are outlined in a simplified manner below.

- http_get: takes a field name as an argument; returns the field value
    - Returns the current value of the field from the AWS IoT device shadow.
- http_post: takes a field name and value as arguments
    - Updates the "field" to the "value" in the AWS IoT device shadow.
- validMove: takes a player, source, and destination as arguments
    - Returns true if the player is allowed to move the piece from the source square to the destination square.
- inCheck: takes a player as an argument
    - Returns true if the player is in check.
- inCheckMate: takes a player as an argument
    - Returns true if the player is in checkmate.

The core behavior of each piece is encoded in the "validMove" function, which differentiates between the piece types. "inCheck" and "inCheckMate" are used to determine if a move is valid, and to correctly determine when the game has ended. The HTTP functions are based off the provided "http_post" function.

User Input

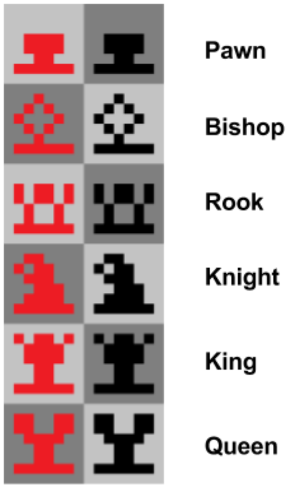
In order to produce a user-friendly product, we opted to use an IR remote for input. The IR remote has a variety of buttons, including arrow keys. However, the setting we used for the remote disabled the use of these keys. Thus, we decided to use the number-pad as the primary means of input. The number pad is shown below.



The numbers ‘2’, ‘4’, ‘6’, and ‘8’ were used for movement (up, left, right, and down respectively). The number ‘5’ was used as the enter key. The ‘last’ button was used to cancel a selection, if the user wants to cancel the current selection and choose a different piece to move. The ‘0’ key is used to restart the game after a game is finished.

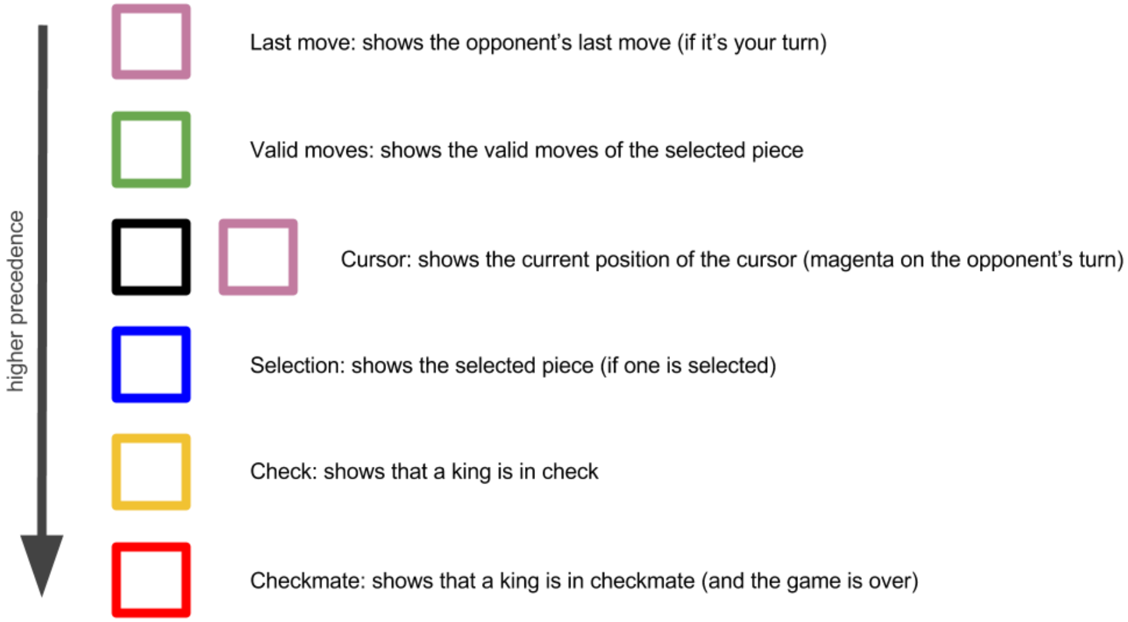
Display

The 128x128 size of the OLED provided a perfect visual space to display the entire chessboard. Examples of the pieces that are displayed to users are shown below.



The “white” player has red pieces, and “black” player has black pieces.

We allow the users to control a cursor on the screen with the IR remote, and select pieces that they would like to move. Throughout this interaction, colored selection-boxes are used to provide feedback to use user. The selection color scheme is shown below. Because the selection boxes outline a single square, they overwrite one another. A precedence was determined to make the experience intuitive for the user.



Last move: shows the opponent’s last move (if it’s your turn)

Valid moves: shows the valid moves of the selected piece

Cursor: shows the current position of the cursor (magenta on the opponent’s turn)

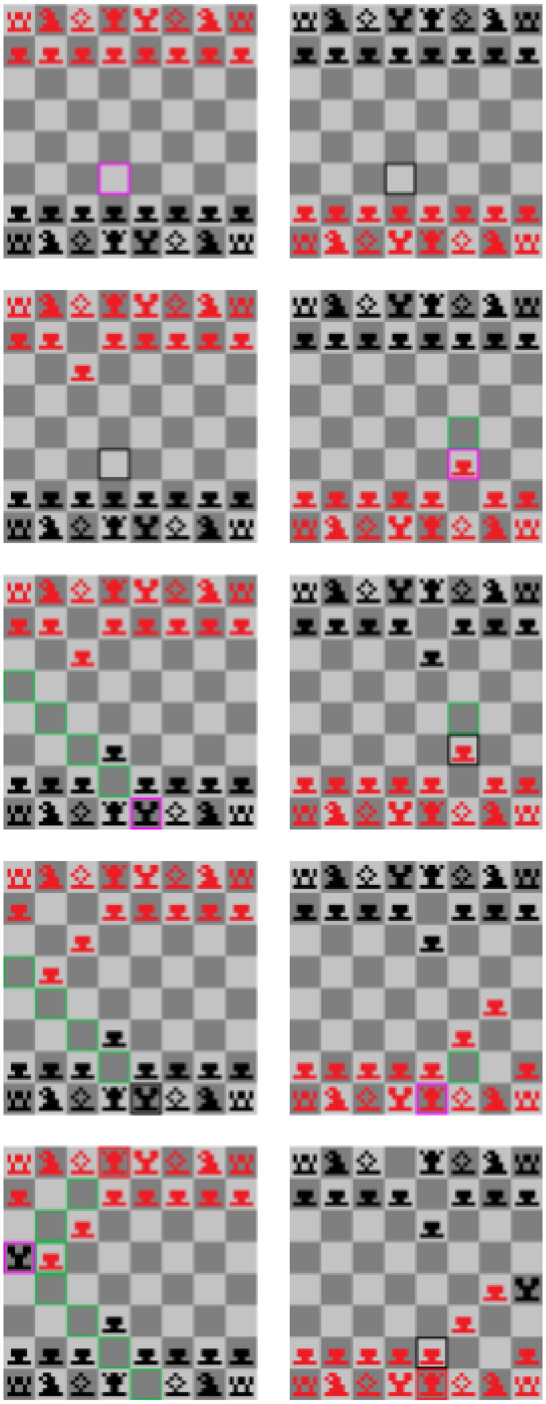
Selection: shows the selected piece (if one is selected)

Check: shows that a king is in check

Checkmate: shows that a king is in checkmate (and the game is over)

Example Game

Below, a full example game is shown below, including each player’s cursor between moves. The point of view of the black player is shown on the left, and the white player is shown on the right.



1. The game starts. It is white player’s turn.
2. White player moves a pawn forward, leaving the cursor on the pawn.
3. Black player moves a pawn forward, leaving the cursor on the black queen.
4. White player moves another pawn forward, leaving the cursor on the red king.
5. Black player moves the queen, leaving the cursor on the black queen.

After the 5th move, the players are notified that that game is over and can press ‘0’ to restart. If both players press ‘0’ the game is restarted.

The messages displayed after the game ends are just one example of the message system we developed. When something important happens, the user is notified by a small blue text box in the upper left corner of the screen. After 2 player input keys are registered by the IR remote, the message disappears. Messages are displayed when: you end your turn, the opponent has played and it is your turn again, you attempt to select a piece when it is the opponent’s turn, or you attempt to move a piece to an invalid position. These messages improve the user experience, and are especially helpful for notifying the player then their opponent has moved (because this can happen at any time, and a user may not notice the moved piece).

Secure Connection

We used AWS IoT in order to communicate between devices. In order to achieve this, we used a single AWS IoT device shadow shared by both boards. This is generally not the intended use for device shadows, but worked well enough for our purposes.

The procedure for creating a secure connection was the same as that used for Lab 5. The root certificate and needed keys were obtained from the AWS portal.

Challenges

We encountered a few challenges while implementing our design. Firstly, getting the boards to get data from the AWS device shadow was difficult. It seemed as though the data received by the HTTP GET command was sometimes old, and would inform a player that it was their turn right after they had just ended their turn. We fixed this issue by calling the http_get function twice, and using the data from the second. It seems as though the first GET flushes the buffer, allowing the next GET to contain up-to-date information.

Another issue we had was with the game logic. At first, it seemed as though there was an intermittent issue where the valid move was not calculated correctly. After closer inspection, we found that this was only caused when a player was in check. In short, the system thought that you putting your opponent in check would negate you being in check from your opponent. This led to the strange situation where both kings could be threatened, but neither player was considered to be in check by the system. It turned out that the “inCheck” function would call the “threatened” function, which would call the “threatens” function, which would call the “validMove” function, which would again call the “inCheck” function. This, somehow, did not crash the program. Instead it manifested itself by returning false for the original inCheck call. We fixed the problem by adding a parameter to “validMove” where one can decide whether to enforce the requirement that a player not be in check after the move, so the function does not call “inCheck” and the cycle is broken.

Future Improvements

Currently, the systems works very well for what we originally intended. However, there are possible improvements that could be made.

First, we currently restart the game when either user reconnects their board. This does not allow for players to begin a game, disconnect the system, then continue the game. In order to achieve this, one would need to store the whole state of the board in AWS in addition to the “turn” and “lastmove” fields. This addition would likely not be difficult, and could allow for players to join an already-started game.

Secondly, we implemented nearly all chess rules, but not all of them. We did not include castling or promotion. Promotion (where one can “upgrade” a pawn when it reaches the other side of the board), would likely be simple to implement. The test games we played, however, did not last very long and only longer games even have the possibility of seeing this mechanic, so we left it out of this version. Castling would also be straightforward to implement, but would require 4 more state variables to be stored locally. Moving either one of your rooks, or your king, disallows castling on a particular side, so information would need to be stored about whether each player is allowed to castle on the left and right. If the functionality of joining partially completed games is also implemented, this information would also have to be stored on AWS.

Lastly, we did not implement draws. In chess, if a player is not in check, but is unable to move any pieces, the game is considered a draw. We did not implement this functionality because of similar reasons as above. This is a very rare situation, that we did not want to put in the time to test. Because of this, however, if this occurs in a game, the game cannot progress or be restarted without resetting the boards. A new function will have to be written to check for this situation; this function would be somewhat similar to the “inCheckMate” function, but will only occur if the player is not in check.

Appendix

1. common.h
    - *WLAN connection code for the CC3200.*
2. main.c
    - *Contains all primary program functionality.*
3. pinmux.c
    - *Pinmux file from Lab 3 was used. Pins assigned for OLED and IR receiver.*

1) common.h

```
//
// Values for below macros shall be modified as per access-point(AP) properties
// SimpleLink device will connect to following AP when application is executed
//
#define SSID_NAME            "eec172"      /* AP SSID */
#define SECURITY_TYPE        SL_SEC_TYPE_OPEN/* Security type (OPEN or WEP or WPA*/
#define SECURITY_KEY         ""            /* Password of the secured AP */
#define SSID_LEN_MAX         32
#define BSSID_LEN_MAX        6


#ifdef NOTERM
#define UART_PRINT(x,...)
#define DBG_PRINT(x,...)
#define ERR_PRINT(x)
#else
#define UART_PRINT Report
#define DBG_PRINT  Report
#define ERR_PRINT(x) Report("Error [%d] at line [%d] in function [%s]
\n\r",x,__LINE__,__FUNCTION__)
#endif

// Loop forever, user can change it as per application's requirement
#define LOOP_FOREVER() \
              {\
              while(1); \
              }

// check the error code and handle it
#define ASSERT_ON_ERROR(error_code)\
              {\
              if(error_code < 0) \
                     {\
                     ERR_PRINT(error_code);\
                     return error_code;\
              }\
              }
```

```
#define SPAWN_TASK_PRIORITY  9
#define SL_STOP_TIMEOUT      200
#define UNUSED(x)            ((x) = (x))
#define SUCCESS              0
#define FAILURE              -1


// Status bits - These are used to set/reset the corresponding bits in
// given variable
typedef enum{
       STATUS_BIT_NWP_INIT = 0, // If this bit is set: Network Processor is
                             // powered up

       STATUS_BIT_CONNECTION,   // If this bit is set: the device is connected to
                             // the AP or client is connected to device (AP)

       STATUS_BIT_IP_LEASED, // If this bit is set: the device has leased IP to
                             // any connected client

       STATUS_BIT_IP_AQUIRED,   // If this bit is set: the device has acquired an IP

       STATUS_BIT_SMARTCONFIG_START, // If this bit is set: the SmartConfiguration
                                    // process is started from SmartConfig app

       STATUS_BIT_P2P_DEV_FOUND,    // If this bit is set: the device (P2P mode)
                             // found any p2p-device in scan

       STATUS_BIT_P2P_REQ_RECEIVED, // If this bit is set: the device (P2P mode)
                             // found any p2p-negotiation request

       STATUS_BIT_CONNECTION_FAILED, // If this bit is set: the device(P2P mode)
                                    // connection to client(or reverse way) is failed

       STATUS_BIT_PING_DONE         // If this bit is set: the device has completed
                             // the ping operation

}e_StatusBits;


#define CLR_STATUS_BIT_ALL(status_variable)  (status_variable = 0)
#define SET_STATUS_BIT(status_variable, bit) status_variable |= (1<<(bit))
#define CLR_STATUS_BIT(status_variable, bit) status_variable &= ~(1<<(bit))
#define CLR_STATUS_BIT_ALL(status_variable)   (status_variable = 0)
#define GET_STATUS_BIT(status_variable, bit) (0 != (status_variable & (1<<(bit))))

#define IS_NW_PROCSR_ON(status_variable)    GET_STATUS_BIT(status_variable,\
                                                  STATUS_BIT_NWP_INIT)
#define IS_CONNECTED(status_variable)       GET_STATUS_BIT(status_variable,\
                                                  STATUS_BIT_CONNECTION)
#define IS_IP_LEASED(status_variable)       GET_STATUS_BIT(status_variable,\
                                                  STATUS_BIT_IP_LEASED)
#define IS_IP_ACQUIRED(status_variable)     GET_STATUS_BIT(status_variable,\
                                                  STATUS_BIT_IP_AQUIRED)
#define IS_SMART_CFG_START(status_variable)  GET_STATUS_BIT(status_variable,\
                                           STATUS_BIT_SMARTCONFIG_START)
#define IS_P2P_DEV_FOUND(status_variable)   GET_STATUS_BIT(status_variable,\
                                                          STATUS_BIT_P2P_DEV_FOUND)
#define IS_P2P_REQ_RCVD(status_variable)    GET_STATUS_BIT(status_variable,\
                                                   STATUS_BIT_P2P_REQ_RECEIVED)
#define IS_CONNECT_FAILED(status_variable)   GET_STATUS_BIT(status_variable,\
                                                   STATUS_BIT_CONNECTION_FAILED)
#define IS_PING_DONE(status_variable)       GET_STATUS_BIT(status_variable,\
                                                          STATUS_BIT_PING_DONE)
```

2) main.c

```
// EEC 172 - Lab 6
// Written by Matthew Martin and Brian Khieu
// June 8, 2017

// Standard includes.
#include <string.h>
#include <stdio.h>

// Driverlib includes.
#include "hw_types.h"
#include "hw_memmap.h"
#include "hw_common_reg.h"
#include "hw_ints.h"
#include "spi.h"
#include "rom.h"
#include "rom_map.h"
#include "utils.h"
#include "prcm.h"
#include "uart.h"
#include "interrupt.h"
#include "gpio.h"
#include "timer.h"

// Common interface includes.
#include "i2c_if.h"
#include "timer_if.h"
#include "pinmux.h"
#include "gpio_if.h"
#include "common.h"
#include "uart_if.h"

// OLED includes.
#include "Adafruit_GFX.h"
#include "Adafruit_SSD1351.h"

// Simplelink includes
#include "simplelink.h"


// Timer base address for the first timer (remote timer).
static volatile unsigned long g_ulBase;
```

```
//*****************************************************************************
//                      MACRO DEFINITIONS
//*****************************************************************************
#define APP_NAME                "Lab6"
#define UART_PRINT              Report
#define FOREVER                 1
#define CONSOLE                 UARTA0_BASE
#define FAILURE                 -1
#define SUCCESS                 0
#define RETERR_IF_TRUE(condition) {if(condition) return FAILURE;}
#define RET_IF_ERR(Func)          {int iRetVal = (Func); \
                                   if (SUCCESS != iRetVal) \
                                     return  iRetVal;}
#define SPI_IF_BIT_RATE  10000000

// Color definitions
#define BLACK           0x0000
#define BLUE            0x001F
#define GREEN           0x07E0
#define CYAN            0x07FF
#define RED             0xF800
#define MAGENTA         0xF81F
#define YELLOW          0xFFE0
#define WHITE           0xFFFF

#define CHESS_COLOR_WHITE   0xF800
#define CHESS_COLOR_BLACK   0x0000
#define CHESS_BOARD_LIGHT   0xFFFF
#define CHESS_BOARD_DARK    0xDEFB  //0xE71C


#define MAX_URI_SIZE 128
#define URI_SIZE MAX_URI_SIZE + 1


#define APPLICATION_NAME        "SSL"
#define APPLICATION_VERSION     "1.1.1.EEC.Winter2017"
#define SERVER_NAME                "adm1p7ok6x5y6.iot.us-east-1.amazonaws.com"
#define GOOGLE_DST_PORT             8443

#define SL_SSL_CA_CERT "/cert/rootCA.der"
#define SL_SSL_PRIVATE "/cert/private.der"
#define SL_SSL_CLIENT  "/cert/client.der"

// Current date.
#define DATE                22    /* Current Date */
#define MONTH               5     /* Month 1-12 */
#define YEAR                2017  /* Current year */
#define HOUR                23    /* Time - hours */
#define MINUTE              39    /* Time - minutes */
#define SECOND              0     /* Time - seconds */

#define GETHEADER "GET /things/CC3200_Thing/shadow HTTP/1.1\n\r"
#define POSTHEADER "POST /things/CC3200_Thing/shadow HTTP/1.1\n\r"
#define HOSTHEADER "Host: adm1p7ok6x5y6.iot.us-east-1.amazonaws.com\r\n"
```

```
#define CHEADER "Connection: Keep-Alive\r\n"
#define CTHEADER "Content-Type: application/json; charset=utf-8\r\n"
#define CLHEADER1 "Content-Length: "
#define CLHEADER2 "\r\n\r\n"

//#define DATA1 "{\"state\": {\n\"desired\" : {\r\n\"default\" : \"Hello Phone\"\n}}}\n\n"
//#define DATA_PREFIX "{\"state\": {\n\"desired\" : {\r\n\"default\" : \""
#define DATA_PREFIX "{\"state\": {\n\"desired\" : {\r\n\""
#define DATA_MIDFIX "\" : \""
#define DATA_POSTFIX "\"\n}}}\n\n"


// Application specific status/error codes
typedef enum{
    // Choosing -0x7D0 to avoid overlap w/ host-driver's error codes
    LAN_CONNECTION_FAILED = -0x7D0,
    INTERNET_CONNECTION_FAILED = LAN_CONNECTION_FAILED - 1,
    DEVICE_NOT_IN_STATION_MODE = INTERNET_CONNECTION_FAILED - 1,

    STATUS_CODE_MAX = -0xBB8
}e_AppStatusCodes;

typedef enum{
    MY_TURN,
    OPPONENT_TURN,
    LARGE_BOARD,
    SMALL_BOARD
}game_mode;

typedef enum{
    BISHOP,
    KING,
    KNIGHT,
    PAWN,
    QUEEN,
    ROOK
}chess_piece;

typedef enum{
    WHITE_PLAYER,
    BLACK_PLAYER,
    NONE_PLAYER
}chess_player;

typedef struct
{
   /* time */
   unsigned long tm_sec;
   unsigned long tm_min;
   unsigned long tm_hour;
   /* date */
   unsigned long tm_day;
   unsigned long tm_mon;
   unsigned long tm_year;
   unsigned long tm_week_day; //not required
   unsigned long tm_year_day; //not required
   unsigned long reserved[3];
}SlDateTime;

typedef struct
{
    char text[32];
    unsigned int fgColor;
    unsigned int bgColor;
    int size;
    int xPos;
    int yPos;
    int aliveCount;
}oledMessage;


// Connection function prototypes.
static long WlanConnect();
static int set_time();
static void BoardInit(void);
static long InitializeAppVariables();
static int tls_connect();
static int connectToAccessPoint();
static int http_post(int, char[], char[]);
static int http_get(int, char*, int, char*, int);

//*** Global variables.
volatile unsigned long  g_ulStatus = 0;//SimpleLink Status
unsigned long  g_ulPingPacketsRecv = 0; //Number of Ping Packets received
unsigned long  g_ulGatewayIP = 0; //Network Gateway IP address
unsigned char  g_ucConnectionSSID[SSID_LEN_MAX+1]; //Connection SSID
unsigned char  g_ucConnectionBSSID[BSSID_LEN_MAX]; //Connection BSSID
signed char    *g_Host = SERVER_NAME;
SlDateTime g_time;
#if defined(ccs) || defined(gcc)
extern void (* const g_pfnVectors[])(void);
#endif
#if defined(ewarm)
extern uVectorEntry __vector_table;
#endif

/*** IR remote global variables***/
// Hold the bit pattern, length, and current position of the last 9 IR remote signals.
// This array is treated as circular. Should only be written to using the addToBitArray
function.
volatile unsigned char bitArray[9];
volatile unsigned int bitArrayLen = 9;
volatile int bitArrayPos = 0;
/*** Internet connection global variables ***/
int socketID = 0;

/*** Chess global variables ***/
// Stores which player this board plays as. Flash each board to be a different player.
chess_player thisPlayer = BLACK_PLAYER;
// Stores the current player's turn. In chess, white player goes first.
chess_player turn = WHITE_PLAYER;
// Stores the winner of the game. NONE_PLAYER if the game is not over.
chess_player winner = NONE_PLAYER;
// Stores the position of the selected piece. (-1, -1) if no selection has been made.
```

```
int selectionX = -1;
int selectionY = -1;
// Stores the current position of the cursor.
int cursorX = 3;
int cursorY = 5;
// Stores the OLED messages.
oledMessage message1;
oledMessage message2;
// Stores the last move data. (-1, -1) and (-1, -1) if there is no last move.
int lastMoveSrcX = -1;
int lastMoveSrcY = -1;
int lastMoveDestX = -1;
int lastMoveDestY = -1;

// Stores the state of the local chess board in character format.
// Black player's pieces are stored as lower case characters, and white player's are stored as
upper case characters.
// Pieces have the following encoding:
//   Bishop: 'b' / 'B'
//     King: 'k' / 'K'
//   Knight: 'n' / 'N'
//     Pawn: 'p' / 'P'
//    Queen: 'q' / 'Q'
//     Rook: 'r' / 'R'
//    Empty: '0'
char chessBoardState[8][8];

// Stores the display shape of each chess piece.
// A '1' indicates that a bit will be displayed in that position.
int bishopArray[6][6] = {
    {0, 0, 1, 0, 0, 0},
    {0, 1, 0, 1, 0, 0},
    {1, 0, 0, 0, 1, 0},
    {0, 1, 0, 1, 0, 0},
    {0, 0, 1, 0, 0, 0},
    {1, 1, 1, 1, 1, 1}
};
int kingArray[6][6] = {
    {1, 0, 1, 1, 0, 1},
    {0, 1, 1, 1, 1, 0},
    {0, 1, 1, 1, 1, 0},
    {0, 0, 1, 1, 0, 0},
    {0, 0, 1, 1, 0, 0},
    {1, 1, 1, 1, 1, 1}
};
int knightArray[6][6] = {
    {0, 1, 1, 0, 0, 0},
    {1, 0, 1, 1, 0, 0},
    {0, 1, 1, 1, 0, 0},
    {0, 0, 1, 1, 1, 0},
    {0, 1, 1, 1, 1, 0},
    {1, 1, 1, 1, 1, 1}
};
int pawnArray[6][6] = {
    {0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0},
    {0, 1, 1, 1, 1, 0},
    {0, 1, 1, 1, 1, 0},
    {0, 0, 1, 1, 0, 0},
    {1, 1, 1, 1, 1, 1}
};
int queenArray[6][6] = {
    {1, 1, 0, 0, 1, 1},
    {1, 1, 0, 0, 1, 1},
    {0, 1, 1, 1, 1, 0},
    {0, 0, 1, 1, 0, 0},
    {0, 0, 1, 1, 0, 0},
    {1, 1, 1, 1, 1, 1}
};
int rookArray[6][6] = {
    {0, 0, 0, 0, 0, 0},
    {1, 0, 1, 1, 0, 1},
    {1, 0, 1, 1, 0, 1},
    {0, 1, 0, 0, 1, 0},
    {0, 1, 0, 0, 1, 0},
    {1, 1, 1, 1, 1, 1}
};

// Chess function prototypes.
int validMove(chess_player, int, int, int, int, char[8][8], int);
int threatened(int, int, char[8][8]);
int threatens(int, int, int, int, char[8][8]);
void doMove(int, int, int, int, char[8][8]);


// Converts the last move data from a character array to ints.
void parseLastMove(char data[], int* xSrc, int* ySrc, int* xDest, int* yDest)
{
    *xSrc = (int)(data[0]) - 48;
    *ySrc = (int)(data[1]) - 48;
    *xDest = (int)(data[2]) - 48;
    *yDest = (int)(data[3]) - 48;
}

// Converts the last move data from ints to a char array.
void createLastMoveData(char data[], int xSrc, int ySrc, int xDest, int yDest)
{
    data[0] = (char)(xSrc + 48);
    data[1] = (char)(ySrc + 48);
    data[2] = (char)(xDest + 48);
    data[3] = (char)(yDest + 48);
}

// Copes the char array chess board from the source board to the destination board.
void copyBoard(char src[8][8], char dest[8][8])
{
    int i, j;
    for (i = 0; i < 8; i++)
    {
        for (j = 0; j < 8; j++)
        {
            dest[i][j] = src[i][j];
        }
    }
```

```
}

// Returns the chess_player representing the opposite of the passed player.
chess_player getOtherPlayer(chess_player player)
{
    if (player == BLACK_PLAYER)
    {
        return WHITE_PLAYER;
    }
    else if (player == WHITE_PLAYER)
    {
        return BLACK_PLAYER;
    }
    else
    {
        return NONE_PLAYER;
    }
}

// Cancels the current piece selection.
void cancelSelection()
{
    selectionX = -1;
    selectionY = -1;
}

// Returns true (1) if there is a piece currently selected.
int selectionActive()
{
    if (selectionX < 0)
    {
        return 0;
    }
    else
    {
        return 1;
    }
}

// Sets the passed board to the state represented by the passed string.
void setChessBoard(char boardString[], char board[8][8])
{
    int i, j;
    int n = 0;
    for (j = 0; j < 8; j++)
    {
        for (i = 0; i < 8; i++)
        {
            board[i][j] = boardString[n];
            n++;
        }
    }
}

// Resets the global chess board to the starting state.
void resetChessBoard()
{
    chessBoardState[0][0] = 'r';
    chessBoardState[1][0] = 'n';
    chessBoardState[2][0] = 'b';
    chessBoardState[3][0] = 'q';
    chessBoardState[4][0] = 'k';
    chessBoardState[5][0] = 'b';
    chessBoardState[6][0] = 'n';
    chessBoardState[7][0] = 'r';

    chessBoardState[0][7] = 'R';
    chessBoardState[1][7] = 'N';
    chessBoardState[2][7] = 'B';
    chessBoardState[3][7] = 'Q';
    chessBoardState[4][7] = 'K';
    chessBoardState[5][7] = 'B';
    chessBoardState[6][7] = 'N';
    chessBoardState[7][7] = 'R';

    int i, j;
    for (i = 0; i < 8; i++)
    {
        chessBoardState[i][1] = 'p';
        chessBoardState[i][6] = 'P';

        for (j = 2; j < 6; j++)
        {
            chessBoardState[i][j] = '0';
        }
    }
}

// Draws the background of the chess board.
void drawChessBoardBackgroundOLED(game_mode mode, chess_player player)
{
    int multi = 8;
    if (mode == LARGE_BOARD)
    {
        multi = 16;
    }

    int i, j;
    for (i = 0; i < 8; i++)
    {
        for (j = 0; j < 8; j++)
        {
            if ((i + j) % 2 == 0)
            {
                fillRect(i * multi, j * multi, multi, multi, CHESS_BOARD_LIGHT);
            }
            else
            {
                fillRect(i * multi, j * multi, multi, multi, CHESS_BOARD_DARK);
            }
        }
    }
}
```

```
// Draws the chess piece of the passed type and player at the passed visual position.
void drawChessPieceOLED(int xPos, int yPos, chess_player player, chess_piece piece, game_mode
mode)
{
    if (xPos < 0 || xPos > 7 || yPos < 0 || yPos > 7) return;

    unsigned int color = CHESS_COLOR_WHITE;
    if (player == BLACK_PLAYER)
    {
        color = CHESS_COLOR_BLACK;
    }

    int i, j;
    if (piece == BISHOP)
    {
        for (i = 0; i < 6; i++)
        {
            for (j = 0; j < 6; j++)
            {
                if (bishopArray[j][i] != 0)
                {
                    if (mode == LARGE_BOARD)
                    {
                        drawPixel((xPos * 16) + ((i + 1) * 2), (yPos * 16) + ((j + 1) * 2),
color);
                        drawPixel((xPos * 16) + ((i + 1) * 2) + 1, (yPos * 16) + ((j + 1) *
2), color);
                        drawPixel((xPos * 16) + ((i + 1) * 2), (yPos * 16) + ((j + 1) * 2) +
1, color);
                        drawPixel((xPos * 16) + ((i + 1) * 2) + 1, (yPos * 16) + ((j + 1) * 2)
+ 1, color);
                    }
                    else
                    {
                        drawPixel((xPos * 8) + i + 1, (yPos * 8) + j + 1, color);
                    }
                }
            }
        }
    }
    else if (piece == KING)
    {
        for (i = 0; i < 6; i++)
        {
            for (j = 0; j < 6; j++)
            {
                if (kingArray[j][i] != 0)
                {
                    if (mode == LARGE_BOARD)
                    {
                        drawPixel((xPos * 16) + ((i + 1) * 2), (yPos * 16) + ((j + 1) * 2),
color);
                        drawPixel((xPos * 16) + ((i + 1) * 2) + 1, (yPos * 16) + ((j + 1) *
2), color);
                        drawPixel((xPos * 16) + ((i + 1) * 2), (yPos * 16) + ((j + 1) * 2) +
1, color);
                        drawPixel((xPos * 16) + ((i + 1) * 2) + 1, (yPos * 16) + ((j + 1) * 2)
+ 1, color);
                    }
                    else
                    {
                        drawPixel((xPos * 8) + i + 1, (yPos * 8) + j + 1, color);
                    }
                }
            }
        }
    }
    else if (piece == KNIGHT)
    {
        for (i = 0; i < 6; i++)
        {
            for (j = 0; j < 6; j++)
            {
                if (knightArray[j][i] != 0)
                {
                    if (mode == LARGE_BOARD)
                    {
                        drawPixel((xPos * 16) + ((i + 1) * 2), (yPos * 16) + ((j + 1) * 2),
color);
                        drawPixel((xPos * 16) + ((i + 1) * 2) + 1, (yPos * 16) + ((j + 1) *
2), color);
                        drawPixel((xPos * 16) + ((i + 1) * 2), (yPos * 16) + ((j + 1) * 2) +
1, color);
                        drawPixel((xPos * 16) + ((i + 1) * 2) + 1, (yPos * 16) + ((j + 1) * 2)
+ 1, color);
                    }
                    else
                    {
                        drawPixel((xPos * 8) + i + 1, (yPos * 8) + j + 1, color);
                    }
                }
            }
        }
    }
    else if (piece == PAWN)
    {
        for (i = 0; i < 6; i++)
        {
            for (j = 0; j < 6; j++)
            {
                if (pawnArray[j][i] != 0)
                {
                    if (mode == LARGE_BOARD)
                    {
                        drawPixel((xPos * 16) + ((i + 1) * 2), (yPos * 16) + ((j + 1) * 2),
color);
                        drawPixel((xPos * 16) + ((i + 1) * 2) + 1, (yPos * 16) + ((j + 1) *
2), color);
                        drawPixel((xPos * 16) + ((i + 1) * 2), (yPos * 16) + ((j + 1) * 2) +
1, color);
                        drawPixel((xPos * 16) + ((i + 1) * 2) + 1, (yPos * 16) + ((j + 1) * 2)
+ 1, color);
                    }
                    else
                    {
                        drawPixel((xPos * 8) + i + 1, (yPos * 8) + j + 1, color);
                    }
                }
            }
        }
    }
    else if (piece == QUEEN)
    {
        for (i = 0; i < 6; i++)
        {
            for (j = 0; j < 6; j++)
            {
                if (queenArray[j][i] != 0)
                {
                    if (mode == LARGE_BOARD)
                    {
                        drawPixel((xPos * 16) + ((i + 1) * 2), (yPos * 16) + ((j + 1) * 2),
color);
                        drawPixel((xPos * 16) + ((i + 1) * 2) + 1, (yPos * 16) + ((j + 1) *
2), color);
                        drawPixel((xPos * 16) + ((i + 1) * 2), (yPos * 16) + ((j + 1) * 2) +
1, color);
                        drawPixel((xPos * 16) + ((i + 1) * 2) + 1, (yPos * 16) + ((j + 1) * 2)
+ 1, color);
                    }
                    else
                    {
                        drawPixel((xPos * 8) + i + 1, (yPos * 8) + j + 1, color);
                    }
                }
            }
        }
    }
    else if (piece == ROOK)
    {
        for (i = 0; i < 6; i++)
        {
            for (j = 0; j < 6; j++)
            {
                if (rookArray[j][i] != 0)
                {
                    if (mode == LARGE_BOARD)
                    {
                        drawPixel((xPos * 16) + ((i + 1) * 2), (yPos * 16) + ((j + 1) * 2),
color);
                        drawPixel((xPos * 16) + ((i + 1) * 2) + 1, (yPos * 16) + ((j + 1) *
2), color);
                        drawPixel((xPos * 16) + ((i + 1) * 2), (yPos * 16) + ((j + 1) * 2) +
1, color);
                        drawPixel((xPos * 16) + ((i + 1) * 2) + 1, (yPos * 16) + ((j + 1) * 2)
+ 1, color);
                    }
                    else
                    {
                        drawPixel((xPos * 8) + i + 1, (yPos * 8) + j + 1, color);
                    }
                }
            }
        }
    }
}

// Returns the chess_player from a piece's char representation.
chess_player getPlayerFromPieceChar(char c)
{
    if (c == 'b' || c == 'k' || c == 'n' || c == 'p' || c == 'q' || c == 'r')
    {
        return BLACK_PLAYER;
    }
    return WHITE_PLAYER;
}

// Returns the chess_piece from a piece's char representation.
chess_piece getPieceFromPieceChar(char c)
{
    if (c == 'b' || c == 'B')
    {
        return BISHOP;
    }
    else if (c == 'k' || c == 'K')
    {
        return KING;
    }
    else if (c == 'n' || c == 'N')
    {
        return KNIGHT;
    }
    else if (c == 'p' || c == 'P')
    {
        return PAWN;
    }
    else if (c == 'q' || c == 'Q')
    {
        return QUEEN;
    }
    else
    {
        return ROOK;
    }
}

// Returns true (1) if the player's king is found on the board.
// If it is found, the king's board position is placed in *x and *y.
int findKing(chess_player player, int* x, int* y, char board[8][8])
{
    char king = 'k';
    if (player == WHITE_PLAYER)
    {
        king = 'K';
    }

    int i, j;
    for (i = 0; i < 8; i++)
    {
        for (j = 0; j < 8; j++)
        {
            if (board[i][j] == king)
            {
                *x = i;
                *y = j;
                return 1;
            }
        }
    }

    return 0;
}

// Returns true if the passed player is in check on the passed board.
int inCheck(chess_player player, char board[8][8])
{
    int x, y;
    if (findKing(player, &x, &y, board))
    {
        if (threatened(x, y, board))
        {
            return 1;
        }
    }

    return 0;
}

// Returns true if the passed player is in check-mate on the passed board.
int inCheckMate(chess_player player, char board[8][8])
{
    // If the player is in check, all possible moves are checked to see if the player can
escape check.
    if (inCheck(player, board))
    {
        int a, b, i, j;
        for (a = 0; a < 8; a++)
        {
            for (b = 0; b < 8; b++)
            {
                if (board[a][b] != '0' && getPlayerFromPieceChar(board[a][b]) == player)
                {
                    for (i = 0; i < 8; i++)
                    {
                        for (j = 0; j < 8; j++)
                        {
                            if (validMove(player, a, b, i, j, board, 1))
                            {
                                char boardCopy[8][8];
                                copyBoard(board, boardCopy);
                                doMove(a, b, i, j, boardCopy);
                                if (!inCheck(player, boardCopy))
                                {
                                    return 0;
                                }
                            }
                        }
                    }
                }
            }
        }
        return 1;
    }
    return 0;
}

// Returns true (1) if the piece at the position is currently "threatened" by the opponent.
int threatened(int x, int y, char board[8][8])
{
    if (board[x][y] == '0')
    {
        return 0;
    }

    int i, j;
    for (i = 0; i < 8; i++)
    {
        for (j = 0; j < 8; j++)
        {
            if (threatens(i, j, x, y, board))
            {
                return 1;
            }
        }
    }
    return 0;
}

// Returns true (1) if a player has a piece at the "From" position that threatens an enemy
piece at the "To" position.
int threatens(int xFrom, int yFrom, int xTo, int yTo, char board[8][8])
{
    // Return false if a board position is invalid.
    if (xFrom < 0 || xFrom >= 8 || yFrom < 0 || yFrom >= 8 || xTo < 0 || xTo >= 8 || yTo < 0
|| yTo >= 8)
    {
        return 0;
    }

    // Returns false if either position is empty.
    if (board[xFrom][yFrom] == '0' || board[xTo][yTo] == '0')
    {
        return 0;
    }

    // Returns false if the pieces belong to the same player.
    if (getPlayerFromPieceChar(board[xFrom][yFrom]) ==
getPlayerFromPieceChar(board[xTo][yTo]))
    {
        return 0;
    }
    chess_player player = getPlayerFromPieceChar(board[xFrom][yFrom]);
    chess_piece piece = getPieceFromPieceChar(board[xFrom][yFrom]);
    if (piece == PAWN)
    {
        int dy = 1;
        if (player == WHITE_PLAYER)
        {
            dy = -1;
        }

        if (yTo - yFrom == dy)
        {
            if (xTo - xFrom == 1 || xTo - xFrom == -1)
            {
                return 1;
            }
        }
        return 0;
    }
    else if (piece == KING)
    {
        if (abs(xTo - xFrom) <= 1 && abs(yTo - yFrom) <= 1)
        {
            return 1;
        }
        return 0;
    }
    else
    {
        return validMove(player, xFrom, yFrom, xTo, yTo, board, 0);
    }
}

// Returns true (1) if the movement of the piece from position 'From' to position 'To' by the player is valid.
// Adds the condition that the player must not be in check after the move, if the useInCheck parameter is true.
int validMove(chess_player player, int xFrom, int yFrom, int xTo, int yTo, char board[8][8], int useInCheck)
{
    // Return false if a board position is invalid.
    if (xFrom < 0 || xFrom >= 8 || yFrom < 0 || yFrom >= 8 || xTo < 0 || xTo >= 8 || yTo < 0 || yTo >= 8)
    {
        return 0;
    }

    // Return false if a player is trying to move a piece that is not their own (or does not exist).
    if (board[xFrom][yFrom] == '0' || getPlayerFromPieceChar(board[xFrom][yFrom]) != player)
    {
        return 0;
    }

    // Return false if a player is trying to move a piece on top of one of their own pieces.
    if (board[xTo][yTo] != '0' && getPlayerFromPieceChar(board[xTo][yTo]) == player)
    {
```

```
    return 0;
}

if (useInCheck)
{
    // Return false if the player is in check after the movement.
    char boardCopy[8][8];
    copyBoard(board, boardCopy);
    doMove(xFrom, yFrom, xTo, yTo, boardCopy);
    if (inCheck(player, boardCopy))
    {
        return 0;
    }
}

chess_piece piece = getPieceFromPieceChar(board[xFrom][yFrom]);
int dx = abs(xTo - xFrom);
int dy = abs(yTo - yFrom);

if (piece == BISHOP)
{
    if (dx == dy)
    {
        int xDir = 1;
        if (xTo < xFrom)
        {
            xDir = -1;
        }
        int yDir = 1;
        if (yTo < yFrom)
        {
            yDir = -1;
        }
        int dist = dx;

        int i;
        for (i = 1; i < dist; i++)
        {
            if (board[xFrom + (i * xDir)][yFrom + (i * yDir)] != '0')
            {
                return 0;
            }
        }
        return 1;
    }
}
else if (piece == KING)
{
    if (dx < 2 && dy < 2)
    {
        return 1;
    }
}
else if (piece == KNIGHT)
{
    if ((dx == 1 && dy == 2) || (dx == 2 && dy == 1))
    {
        return 1;
    }
}
else if (piece == PAWN)
{
    if ((player == WHITE_PLAYER && yTo < yFrom) || (player == BLACK_PLAYER && yTo > yFrom))
    { // The white player may only move pawns up. Black may only move pawns down.
        if (dy == 1)
        {
            char atTo = board[xTo][yTo];
            if ((atTo == '0' && dx == 0) || (atTo != '0' && dx == 1))
            {
                // Players can move their pawn one space forward if the space is empty.
                // They can move a pawn one space forward and left or right if the space contains an opponent's piece.
                return 1;
            }

        }
        else if (dy == 2)
        {
            int yDir = 1;
            if (player == WHITE_PLAYER)
            {
                yDir = -1;
            }
            if (board[xFrom][yFrom + yDir] == '0' && board[xFrom][yFrom + (2 * yDir)] == '0' && dx == 0)
            {
                if ((player == WHITE_PLAYER && yFrom == 6) || (player == BLACK_PLAYER && yFrom == 1))
                {
                    // Players can move a pawn from their own second row two spaces forward,
                    // if both spaces in front of it are clear.
                    return 1;
                }
            }
        }
    }
}
else if (piece == QUEEN)
{
    if (dx == 0 || dy == 0 || dx == dy)
    {
        int xDir = 0;
        if (xTo < xFrom)
        {
            xDir = -1;
        }
        else if (xTo > xFrom)
        {
            xDir = 1;
        }

        int yDir = 0;
        if (yTo < yFrom)
        {
            yDir = -1;
        } else if (yTo > yFrom)
        {
            yDir = 1;
        }

        int dist = dx;
        if (dx == 0)
        {
            dist = dy;
        }

        int i;
        for (i = 1; i < dist; i++)
        {
            if (board[xFrom + (i * xDir)][yFrom + (i * yDir)] != '0')
            {
                return 0;
            }
        }
        return 1;
    }
}
else if (piece == ROOK)
{
    if (dx == 0 || dy == 0)
    { // Rooks may only move in one direction at a time.
        int xDir = 0;
        if (xTo < xFrom)
        {
            xDir = -1;
        }
        else if (xTo > xFrom)
        {
            xDir = 1;
        }

        int yDir = 0;
        if (yTo < yFrom)
        {
            yDir = -1;
        } else if (yTo > yFrom)
        {
            yDir = 1;
        }

        int dist = dx;
        if (dx == 0)
        {
            dist = dy;
        }

        int i;
        for (i = 1; i < dist; i++)
        {
            if (board[xFrom + (i * xDir)][yFrom + (i * yDir)] != '0')
            {
                return 0;
            }
        }
        return 1;
    }
}

return 0;
}

// Moves a chess piece. Assumes the move has already been checked for validity.
void doMove(int xFrom, int yFrom, int xTo, int yTo, char board[8][8])
{
char c = board[xFrom][yFrom];
board[xFrom][yFrom] = '0';
board[xTo][yTo] = c;
}

// Draws the chess board from the passed player's perspective.
void drawChessBoardOLED(game_mode mode, chess_player player)
{
drawChessBoardBackgroundOLED(mode, player);

int i, j;
for (i = 0; i < 8; i++)
{
    for (j = 0; j < 8; j++)
    {
        char c = chessBoardState[i][j];
        if (c != '0')
        {
            if (player == WHITE_PLAYER)
            {
                drawChessPieceOLED(i, j, getPlayerFromPieceChar(c),
getPieceFromPieceChar(c), mode);
            }
            else
            {
                drawChessPieceOLED(7 - i, 7 - j, getPlayerFromPieceChar(c),
getPieceFromPieceChar(c), mode);
            }
        }
    }
}
}

// Draws a cursor at the specified board position, from the point of view of the specified
player, in the specified color.
void drawCursorOLED(int x, int y, game_mode mode, unsigned int color, chess_player player)
{
if (x < 0 || x >= 8 || y < 0 || y >= 8)
{
    return;
}
int size = 8;
if (mode == LARGE_BOARD)
{
    size = 16;
}

if (player == WHITE_PLAYER)
{
    drawRect(x * size, y * size, size, size, color);
}
else
{
    drawRect((7 - x) * size, (7 - y) * size, size, size, color);
}
}

// Draws the valid moves for the currently selected, or cursor-selected, piece.
void drawValidMovesOLED(game_mode mode, chess_player player)
{
int x = cursorX;
int y = cursorY;
if (selectionActive())
{
    x = selectionX;
    y = selectionY;
}

if (chessBoardState[x][y] != '0')
{
    int i, j;
    for (i = 0; i < 8; i++)
    {
        for (j = 0; j < 8; j++)
        {
            if (validMove(player, x, y, i, j, chessBoardState, 1))
            {
                drawCursorOLED(i, j, mode, GREEN, player);
            }
        }
    }
}
}


// Clears the bit array (filling it with 1's).
void clearBitArray()
{
int i;
for (i = 0; i < bitArrayLen; i++)
{
    bitArray[i] = 1;
}
}

// Returns true if-and-only-if no button has recently been pressed on the IR remote,
// and thus the bit-array is "empty" (filled with 1's).
unsigned int isBitArrayClear()
```

```
{
    int i;
    for (i = 0; i < bitArrayLen; i++)
    {
        if (bitArray[i] == 0)
        {
            return 0;
        }
    }
    return 1;
}

// Returns the button that was pressed, based on the information currently in the bit-array.
char readBitArray()
{
    unsigned char bitArrayCopy[9];

    // Copy the bit-array into a copy. Because the original bit-array is circular,
    // it is copied such that the most recent bit is at the beginning of the copy.
    int posCopy = bitArrayPos - 1;
    if (posCopy < 0)
    {
        posCopy = bitArrayLen - 1;
    }
    int i;
    for (i = 0; i < bitArrayLen; i++)
    {
        posCopy++;
        if (posCopy >= bitArrayLen)
        {
            posCopy = 0;
        }
        bitArrayCopy[i] = bitArray[posCopy];
    }

    // Once the array is read, it is cleared to ensure that each button press is only read
once.
    clearBitArray();

    // Decode the bit-array to return which button was pressed.
    // Recognizes only the number buttons '0' through '1', the mute button, and the last
button.
    // All other buttons on the remote (including no button recently pressed), are returned as
the special character 'X'.
    char result = 'X';
    if (bitArrayCopy[0] != 0 && bitArrayCopy[6] == 0 && bitArrayCopy[7] == 0 &&
bitArrayCopy[8] == 0)
    {
        if (bitArrayCopy[1] != 0 && bitArrayCopy[2] == 0 && bitArrayCopy[3] == 0 &&
bitArrayCopy[4] == 0 && bitArrayCopy[5] == 0)
        {
            result = '1';
        }
        else if (bitArrayCopy[1] == 0 && bitArrayCopy[2] != 0 && bitArrayCopy[3] == 0 &&
bitArrayCopy[4] == 0 && bitArrayCopy[5] == 0)
        {
            result = '2';
        }
        else if (bitArrayCopy[1] != 0 && bitArrayCopy[2] != 0 && bitArrayCopy[3] == 0 &&
bitArrayCopy[4] == 0 && bitArrayCopy[5] == 0)
        {
            result = '3';
        }
        else if (bitArrayCopy[1] == 0 && bitArrayCopy[2] == 0 && bitArrayCopy[3] != 0 &&
bitArrayCopy[4] == 0 && bitArrayCopy[5] == 0)
        {
            result = '4';
        }
        else if (bitArrayCopy[1] != 0 && bitArrayCopy[2] == 0 && bitArrayCopy[3] != 0 &&
bitArrayCopy[4] == 0 && bitArrayCopy[5] == 0)
        {
            result = '5';
        }
        else if (bitArrayCopy[1] == 0 && bitArrayCopy[2] != 0 && bitArrayCopy[3] != 0 &&
bitArrayCopy[4] == 0 && bitArrayCopy[5] == 0)
        {
            result = '6';
        }
        else if (bitArrayCopy[1] != 0 && bitArrayCopy[2] != 0 && bitArrayCopy[3] != 0 &&
bitArrayCopy[4] == 0 && bitArrayCopy[5] == 0)
        {
            result = '7';
        }
        else if (bitArrayCopy[1] == 0 && bitArrayCopy[2] == 0 && bitArrayCopy[3] == 0 &&
bitArrayCopy[4] != 0 && bitArrayCopy[5] == 0)
        {
            result = '8';
        }
        else if (bitArrayCopy[1] != 0 && bitArrayCopy[2] == 0 && bitArrayCopy[3] == 0 &&
bitArrayCopy[4] != 0 && bitArrayCopy[5] == 0)
        {
            result = '9';
        }
        else if (bitArrayCopy[1] == 0 && bitArrayCopy[2] == 0 && bitArrayCopy[3] == 0 &&
bitArrayCopy[4] == 0 && bitArrayCopy[5] == 0)
        {
            result = '0';
        }
        else if (bitArrayCopy[1] == 0 && bitArrayCopy[2] != 0 && bitArrayCopy[3] == 0 &&
bitArrayCopy[4] == 0 && bitArrayCopy[5] != 0)
        {
            result = 'M';
        }
        else if (bitArrayCopy[1] == 0 && bitArrayCopy[2] == 0 && bitArrayCopy[3] != 0 &&
bitArrayCopy[4] != 0 && bitArrayCopy[5] != 0)
        {
            result = 'L';
        }
    }

    return result;
}

// Adds the passed bit to the circular bit-array.
```

```
void addToBitArray(unsigned char val)
{
    bitArray[bitArrayPos] = val;
    bitArrayPos++;
    if (bitArrayPos >= bitArrayLen)
    {
        bitArrayPos = 0;
    }
}

void redrawOLED(chess_player player)
{
    // Assign cursor colors.
    unsigned int cursorColor = BLACK;
    if (turn != thisPlayer)
    {
        cursorColor = MAGENTA;
    }
    unsigned int selectColor = BLUE;
    unsigned int checkColor = YELLOW;
    unsigned int checkMateColor = RED;
    unsigned int lastMoveColor = MAGENTA;

    // Render the board.
    drawChessBoardOLED(LARGE_BOARD, player);
    // Render the cursor boxes.
    if (turn == thisPlayer)
    {
        drawCursorOLED(lastMoveSrcX, lastMoveSrcY, LARGE_BOARD, lastMoveColor, player);
        drawCursorOLED(lastMoveDestX, lastMoveDestY, LARGE_BOARD, lastMoveColor, player);
    }
    drawValidMovesOLED(LARGE_BOARD, player);
    drawCursorOLED(cursorX, cursorY, LARGE_BOARD, cursorColor, player);
    drawCursorOLED(selectionX, selectionY, LARGE_BOARD, selectColor, player);

    // Draw the white player's king check box.
    if (inCheck(WHITE_PLAYER, chessBoardState))
    {
        int wx = -1;
        int wy = -1;
        findKing(WHITE_PLAYER, &wx, &wy, chessBoardState);
        if (winner == BLACK_PLAYER)
        {
            drawCursorOLED(wx, wy, LARGE_BOARD, checkMateColor, player);
        }
        else
        {
            drawCursorOLED(wx, wy, LARGE_BOARD, checkColor, player);
        }
    }

    // Draw the black player's king check box.
    if (inCheck(BLACK_PLAYER, chessBoardState))
    {
        int bx = -1;
        int by = -1;
        findKing(BLACK_PLAYER, &bx, &by, chessBoardState);
        if (winner == WHITE_PLAYER)
        {
            drawCursorOLED(bx, by, LARGE_BOARD, checkMateColor, player);
        }
        else
        {
            drawCursorOLED(bx, by, LARGE_BOARD, checkColor, player);
        }
    }

    // Draw the messages.
    if (message1.aliveCount > 0)
    {
        setTextColor(message1.fgColor, message1.bgColor);
        setTextSize(message1.size);
        setCursor(message1.xPos, message1.yPos);
        Outstr(message1.text);
    }
    if (message2.aliveCount > 0)
    {
        setTextColor(message2.fgColor, message2.bgColor);
        setTextSize(message2.size);
        setCursor(message2.xPos, message2.yPos);
        Outstr(message2.text);
    }
}


// Decodes the button pressed into a command.
char getCommandFromButton(char button)
{
    if (button == '4')
    { // left
        return 'L';
    }
    else if (button == '6')
    { // right
        return 'R';
    }
    else if (button == '2')
    { // up
        return 'U';
    }
    else if (button == '8')
    { // down
        return 'D';
    }
    else if (button == 'L')
    { // cancel
        return 'C';
    }
    else if (button == '5')
    { // enter
        return 'E';
    }
    else if (button == '0')
    { // restart
```

```
        return 'S';
    }
    else
    {
        return 0;
    }
}


// Executes the command based on the button that was pressed to produce it.
// Called by the remote timer expiration handler.
void doCommand(char command, char button)
{
    int redraw = 0;

    if (command == 'L') // left
    {
        if (thisPlayer == WHITE_PLAYER)
        {
            cursorX--;
        }
        else
        {
            cursorX++;
        }

        if (cursorX < 0)
        {
            cursorX = 7;
        }
        else if (cursorX > 7)
        {
            cursorX = 0;
        }
        redraw = 1;
    }
    else if (command == 'R') // right
    {
        if (thisPlayer == WHITE_PLAYER)
        {
            cursorX++;
        }
        else
        {
            cursorX--;
        }

        if (cursorX < 0)
        {
            cursorX = 7;
        }
        else if (cursorX > 7)
        {
            cursorX = 0;
        }
        redraw = 1;
    }
```

```
else if (command == 'U') // up
{
    if (thisPlayer == WHITE_PLAYER)
    {
        cursorY--;
    }
    else
    {
        cursorY++;
    }

    if (cursorY < 0)
    {
        cursorY = 7;
    }
    else if (cursorY > 7)
    {
        cursorY = 0;
    }
    redraw = 1;
}
else if (command == 'D') // down
{
    if (thisPlayer == WHITE_PLAYER)
    {
        cursorY++;
    }
    else
    {
        cursorY--;
    }

    if (cursorY < 0)
    {
        cursorY = 7;
    }
    else if (cursorY > 7)
    {
        cursorY = 0;
    }
    redraw = 1;
}
else if (command == 'E') // enter
{
    if (selectionActive())
    {
        chess_player p = getPlayerFromPieceChar(chessBoardState[selectionX][selectionY]);
        if (validMove(p, selectionX, selectionY, cursorX, cursorY, chessBoardState, 1))
        {
            strcpy(message1.text, "Wait while your");
            message1.fgColor = WHITE;
            message1.bgColor = BLUE;
            message1.size = 1;
            message1.xPos = 10;
            message1.yPos = 10;
            message1.aliveCount = 2;
            strcpy(message2.text, "opponent moves");
            message2.fgColor = WHITE;
            message2.bgColor = BLUE;
            message2.size = 1;
            message2.xPos = 10;
            message2.yPos = 18;
            message2.aliveCount = 2;

            doMove(selectionX, selectionY, cursorX, cursorY, chessBoardState);
            if (inCheckMate(getOtherPlayer(thisPlayer), chessBoardState))
            {
                winner = thisPlayer;

                strcpy(message1.text, "You win!");
                message1.fgColor = WHITE;
                message1.bgColor = BLUE;
                message1.size = 2;
                message1.xPos = 16;
                message1.yPos = 40;
                message1.aliveCount = 2;

                strcpy(message2.text, "Press 0 to restart");
                message2.fgColor = WHITE;
                message2.bgColor = BLUE;
                message2.size = 1;
                message2.xPos = 10;
                message2.yPos = 60;
                message2.aliveCount = 2;
            }

            // Switch the state to the other player.
            turn = getOtherPlayer(turn);

            // Update AWS with the last move data.
            char lastMoveData[5] = "xxxx";
            createLastMoveData(lastMoveData, selectionX, selectionY, cursorX, cursorY);
            http_post(socketID, "lastmove", lastMoveData);

            // Update AWS with the turn data.
            char turnData[2] = "B";
            if (thisPlayer == BLACK_PLAYER)
            {
                strcpy(turnData, "W");
            }
            http_post(socketID, "turn", turnData);

            // Deselect the moved piece.
            cancelSelection();
        }
        else
        { // The move is not valid.
            strcpy(message1.text, "Invalid move");
            message1.fgColor = WHITE;
            message1.bgColor = BLUE;
            message1.size = 1;
            message1.xPos = 10;
            message1.yPos = 10;
```

```
            message1.aliveCount = 2;
            message2.aliveCount = 0;
        }
    }
    else
    { // No piece is currently selected.
        if (chessBoardState[cursorX][cursorY] != '0' &&
            getPlayerFromPieceChar(chessBoardState[cursorX][cursorY]) == thisPlayer)
        { // The cursor is on a friendly piece.
            if (thisPlayer == turn)
            {
                selectionX = cursorX;
                selectionY = cursorY;
            }
            else
            {
                strcpy(message1.text, "It's not your turn");
                message1.fgColor = WHITE;
                message1.bgColor = BLUE;
                message1.size = 1;
                message1.xPos = 10;
                message1.yPos = 10;
                message1.aliveCount = 2;
                message2.aliveCount = 0;
            }
        }
    }
    redraw = 1;
}
else if (command == 'C') // cancel
{
    cancelSelection();
    redraw = 1;
}
else if (command == 'S') // restart
{
    if (winner != NONE_PLAYER)
    {
        winner = NONE_PLAYER;

        http_post(socketID, "turn", "W");
        http_post(socketID, "lastmove", "xxxx");
        resetChessBoard();
        turn = WHITE_PLAYER;

        lastMoveSrcX = -1;
        lastMoveSrcY = -1;
        lastMoveDestX = -1;
        lastMoveDestY = -1;

        redraw = 1;

        if (thisPlayer == WHITE_PLAYER)
        {
            strcpy(message1.text, "Your turn");
            message1.fgColor = WHITE;
            message1.bgColor = BLUE;
```

```
                message1.size = 1;
                message1.xPos = 10;
                message1.yPos = 10;
                message1.aliveCount = 2;
                message2.aliveCount = 0;
            }
            else
            {
                strcpy(message1.text, "Wait while your");
                message1.fgColor = WHITE;
                message1.bgColor = BLUE;
                message1.size = 1;
                message1.xPos = 10;
                message1.yPos = 10;
                message1.aliveCount = 2;

                strcpy(message2.text, "opponent moves");
                message2.fgColor = WHITE;
                message2.bgColor = BLUE;
                message2.size = 1;
                message2.xPos = 10;
                message2.yPos = 18;
                message2.aliveCount = 2;
            }
        }
    }

    // Redraw the board if something worthwhile happened.
    if (redraw)
    {
        redrawOLED(thisPlayer);
        // Decrement the message alive counts.
        message1.aliveCount--;
        message2.aliveCount--;
    }
}


// Remote timer expiration handler.
// This timer will expire when a full button-press has been registered and the user releases the button.
// This timer will also expire if too much time has gone by where no buttons are pressed
//     - in this case a special "buttonPressed" value is read to indicate that no button has been pressed.
void TimerBaseIntHandler(void)
{
    // Clear this timer's interrupt.
    Timer_IF_InterruptClear(g_ulBase);

    // Find out which button has been pressed.
    char buttonPressed = readBitArray();
    // Determine which command to execute based on the pressed button.
    char command = getCommandFromButton(buttonPressed);
    // Execute the command based on which button was pressed.
    doCommand(command, buttonPressed);
}
```

```
// IR Remote interrupt handler.
// Called when the GPIO input pin (from the IR receiver) goes low.
static void IRRemoteInterruptHandler(void)
{
    // Get the time since the last IR remote interrupt.
    // If it has been a long time, treat this "bit" as a 1, otherwise treat it as a 0.
    unsigned long timeElapsed = TimerValueGet(g_ulBase, TIMER_A);
    unsigned char bit = (timeElapsed > 100000) ? 1 : 0;

    // Push the most recent bit into the circular bit-array.
    addToBitArray(bit);

    // Restart both the IR remote timer, and the idle-timeout timer.
    TimerValueSet(g_ulBase, TIMER_A, 0);

    // Clear IR remote interrupts.
    unsigned long ulStatus = GPIOIntStatus (GPIOA1_BASE, true);
    GPIOIntClear(GPIOA1_BASE, ulStatus);
}


//*****************************************************************************
// SimpleLink Asynchronous Event Handlers -- Start
//*****************************************************************************

//*****************************************************************************
//
//! \brief The Function Handles WLAN Events
//!
//! \param[in]  pWlanEvent - Pointer to WLAN Event Info
//!
//! \return None
//!
//*****************************************************************************
void SimpleLinkWlanEventHandler(SlWlanEvent_t *pWlanEvent) {
    if(!pWlanEvent) {
        return;
    }

    switch(pWlanEvent->Event) {
        case SL_WLAN_CONNECT_EVENT: {
            SET_STATUS_BIT(g_ulStatus, STATUS_BIT_CONNECTION);

            //
            // Information about the connected AP (like name, MAC etc) will be
            // available in 'slWlanConnectAsyncResponse_t'.
            // Applications can use it if required
            //
            //  slWlanConnectAsyncResponse_t *pEventData = NULL;
            // pEventData = &pWlanEvent->EventData.STAandP2PModeWlanConnected;
            //

            // Copy new connection SSID and BSSID to global parameters
            memcpy(g_ucConnectionSSID,pWlanEvent->EventData.
                   STAandP2PModeWlanConnected.ssid_name,
                   pWlanEvent->EventData.STAandP2PModeWlanConnected.ssid_len);
            memcpy(g_ucConnectionBSSID,
                   pWlanEvent->EventData.STAandP2PModeWlanConnected.bssid,
                   SL_BSSID_LENGTH);

            UART_PRINT("[WLAN EVENT] STA Connected to the AP: %s , "
                       "BSSID: %x:%x:%x:%x:%x:%x\n\r",
                       g_ucConnectionSSID,g_ucConnectionBSSID[0],
                       g_ucConnectionBSSID[1],g_ucConnectionBSSID[2],
                       g_ucConnectionBSSID[3],g_ucConnectionBSSID[4],
                       g_ucConnectionBSSID[5]);
        }
        break;

        case SL_WLAN_DISCONNECT_EVENT: {
            slWlanConnectAsyncResponse_t*  pEventData = NULL;

            CLR_STATUS_BIT(g_ulStatus, STATUS_BIT_CONNECTION);
            CLR_STATUS_BIT(g_ulStatus, STATUS_BIT_IP_AQUIRED);

            pEventData = &pWlanEvent->EventData.STAandP2PModeDisconnected;

            // If the user has initiated 'Disconnect' request,
            //'reason_code' is SL_USER_INITIATED_DISCONNECTION
            if(SL_USER_INITIATED_DISCONNECTION == pEventData->reason_code) {
                UART_PRINT("[WLAN EVENT]Device disconnected from the AP: %s,"
                    "BSSID: %x:%x:%x:%x:%x:%x on application's request \n\r",
                           g_ucConnectionSSID,g_ucConnectionBSSID[0],
                           g_ucConnectionBSSID[1],g_ucConnectionBSSID[2],
                           g_ucConnectionBSSID[3],g_ucConnectionBSSID[4],
                           g_ucConnectionBSSID[5]);
            }
            else {
                UART_PRINT("[WLAN ERROR]Device disconnected from the AP AP: %s, "
                           "BSSID: %x:%x:%x:%x:%x:%x on an ERROR..!! \n\r",
                           g_ucConnectionSSID,g_ucConnectionBSSID[0],
                           g_ucConnectionBSSID[1],g_ucConnectionBSSID[2],
                           g_ucConnectionBSSID[3],g_ucConnectionBSSID[4],
                           g_ucConnectionBSSID[5]);
            }
            memset(g_ucConnectionSSID,0,sizeof(g_ucConnectionSSID));
            memset(g_ucConnectionBSSID,0,sizeof(g_ucConnectionBSSID));
        }
        break;

        default: {
            UART_PRINT("[WLAN EVENT] Unexpected event [0x%x]\n\r",
                       pWlanEvent->Event);
        }
        break;
    }
}

//*****************************************************************************
//
//! \brief This function handles network events such as IP acquisition, IP
//!           leased, IP released etc.
//!
```

```
//! \param[in]  pNetAppEvent - Pointer to NetApp Event Info
//!
//! \return None
//!
//*****************************************************************************
void SimpleLinkNetAppEventHandler(SlNetAppEvent_t *pNetAppEvent) {
    if(!pNetAppEvent) {
        return;
    }

    switch(pNetAppEvent->Event) {
        case SL_NETAPP_IPV4_IPACQUIRED_EVENT: {
            SlIpV4AcquiredAsync_t *pEventData = NULL;

            SET_STATUS_BIT(g_ulStatus, STATUS_BIT_IP_AQUIRED);

            //Ip Acquired Event Data
            pEventData = &pNetAppEvent->EventData.ipAcquiredV4;

            //Gateway IP address
            g_ulGatewayIP = pEventData->gateway;

            UART_PRINT("[NETAPP EVENT] IP Acquired: IP=%d.%d.%d.%d , "
                       "Gateway=%d.%d.%d.%d\n\r",
            SL_IPV4_BYTE(pNetAppEvent->EventData.ipAcquiredV4.ip,3),
            SL_IPV4_BYTE(pNetAppEvent->EventData.ipAcquiredV4.ip,2),
            SL_IPV4_BYTE(pNetAppEvent->EventData.ipAcquiredV4.ip,1),
            SL_IPV4_BYTE(pNetAppEvent->EventData.ipAcquiredV4.ip,0),
            SL_IPV4_BYTE(pNetAppEvent->EventData.ipAcquiredV4.gateway,3),
            SL_IPV4_BYTE(pNetAppEvent->EventData.ipAcquiredV4.gateway,2),
            SL_IPV4_BYTE(pNetAppEvent->EventData.ipAcquiredV4.gateway,1),
            SL_IPV4_BYTE(pNetAppEvent->EventData.ipAcquiredV4.gateway,0));
        }
        break;

        default: {
            UART_PRINT("[NETAPP EVENT] Unexpected event [0x%x] \n\r",
                       pNetAppEvent->Event);
        }
        break;
    }
}

//*****************************************************************************
//
//! \brief This function handles HTTP server events
//!
//! \param[in]  pServerEvent - Contains the relevant event information
//! \param[in]    pServerResponse - Should be filled by the user with the
//!                                      relevant response information
//!
//! \return None
//!
//****************************************************************************
void SimpleLinkHttpServerCallback(SlHttpServerEvent_t *pHttpEvent, SlHttpServerResponse_t
*pHttpResponse) {
    // Unused in this application
```

```
}

//*****************************************************************************
//
//! \brief This function handles General Events
//!
//! \param[in]     pDevEvent - Pointer to General Event Info
//!
//! \return None
//!
//*****************************************************************************
void SimpleLinkGeneralEventHandler(SlDeviceEvent_t *pDevEvent) {
    if(!pDevEvent) {
        return;
    }

    //
    // Most of the general errors are not FATAL are are to be handled
    // appropriately by the application
    //
    UART_PRINT("[GENERAL EVENT] - ID=[%d] Sender=[%d]\n\n",
               pDevEvent->EventData.deviceEvent.status,
               pDevEvent->EventData.deviceEvent.sender);
}

//*****************************************************************************
//
//! This function handles socket events indication
//!
//! \param[in]      pSock - Pointer to Socket Event Info
//!
//! \return None
//!
//*****************************************************************************
void SimpleLinkSockEventHandler(SlSockEvent_t *pSock) {
    if(!pSock) {
        return;
    }

    switch( pSock->Event ) {
        case SL_SOCKET_TX_FAILED_EVENT:
            switch( pSock->socketAsyncEvent.SockTxFailData.status) {
                case SL_ECLOSE:
                    UART_PRINT("[SOCK ERROR] - close socket (%d) operation "
                                "failed to transmit all queued packets\n\n",
                                    pSock->socketAsyncEvent.SockTxFailData.sd);
                    break;
                default:
                    UART_PRINT("[SOCK ERROR] - TX FAILED  :  socket %d , reason "
                                "(%d) \n\n",
                                pSock->socketAsyncEvent.SockTxFailData.sd,
pSock->socketAsyncEvent.SockTxFailData.status);
                  break;
            }
            break;

        default:
            UART_PRINT("[SOCK EVENT] - Unexpected Event [%x0x]\n\n",pSock->Event);
          break;
    }
}
//*****************************************************************************
// SimpleLink Asynchronous Event Handlers -- End
//*****************************************************************************


//*****************************************************************************
//
//! \brief This function initializes the application variables
//!
//! \param    0 on success else error code
//!
//! \return None
//!
//*****************************************************************************
static long InitializeAppVariables() {
    g_ulStatus = 0;
    g_ulGatewayIP = 0;
    g_Host = SERVER_NAME;
    memset(g_ucConnectionSSID,0,sizeof(g_ucConnectionSSID));
    memset(g_ucConnectionBSSID,0,sizeof(g_ucConnectionBSSID));
    return SUCCESS;
}


//*****************************************************************************
//! \brief This function puts the device in its default state. It:
//!           - Set the mode to STATION
//!           - Configures connection policy to Auto and AutoSmartConfig
//!           - Deletes all the stored profiles
//!           - Enables DHCP
//!           - Disables Scan policy
//!           - Sets Tx power to maximum
//!           - Sets power policy to normal
//!           - Unregister mDNS services
//!           - Remove all filters
//!
//! \param   none
//! \return  On success, zero is returned. On error, negative is returned
//*****************************************************************************
static long ConfigureSimpleLinkToDefaultState() {
    SlVersionFull   ver = {0};
    _WlanRxFilterOperationCommandBuff_t  RxFilterIdMask = {0};

    unsigned char ucVal = 1;
    unsigned char ucConfigOpt = 0;
    unsigned char ucConfigLen = 0;
    unsigned char ucPower = 0;

    long lRetVal = -1;
    long lMode = -1;

    lMode = sl_Start(0, 0, 0);
    ASSERT_ON_ERROR(lMode);
    // If the device is not in station-mode, try configuring it in station-mode
    if (ROLE_STA != lMode) {
        if (ROLE_AP == lMode) {
            // If the device is in AP mode, we need to wait for this event
            // before doing anything
            while(!IS_IP_ACQUIRED(g_ulStatus)) {
#ifndef SL_PLATFORM_MULTI_THREADED
              _SlNonOsMainLoopTask();
#endif
            }
        }

        // Switch to STA role and restart
        lRetVal = sl_WlanSetMode(ROLE_STA);
        ASSERT_ON_ERROR(lRetVal);

        lRetVal = sl_Stop(0xFF);
        ASSERT_ON_ERROR(lRetVal);

        lRetVal = sl_Start(0, 0, 0);
        ASSERT_ON_ERROR(lRetVal);

        // Check if the device is in station again
        if (ROLE_STA != lRetVal) {
            // We don't want to proceed if the device is not coming up in STA-mode
            return DEVICE_NOT_IN_STATION_MODE;
        }
    }

    // Get the device's version-information
    ucConfigOpt = SL_DEVICE_GENERAL_VERSION;
    ucConfigLen = sizeof(ver);
    lRetVal = sl_DevGet(SL_DEVICE_GENERAL_CONFIGURATION, &ucConfigOpt,
                                &ucConfigLen, (unsigned char *)(&ver));
    ASSERT_ON_ERROR(lRetVal);

    UART_PRINT("Host Driver Version: %s\n\r",SL_DRIVER_VERSION);
    UART_PRINT("Build Version %d.%d.%d.%d.31.%d.%d.%d.%d.%d.%d.%d.%d\n\r",
    ver.NwpVersion[0],ver.NwpVersion[1],ver.NwpVersion[2],ver.NwpVersion[3],
    ver.ChipFwAndPhyVersion.FwVersion[0],ver.ChipFwAndPhyVersion.FwVersion[1],
    ver.ChipFwAndPhyVersion.FwVersion[2],ver.ChipFwAndPhyVersion.FwVersion[3],
    ver.ChipFwAndPhyVersion.PhyVersion[0],ver.ChipFwAndPhyVersion.PhyVersion[1],
    ver.ChipFwAndPhyVersion.PhyVersion[2],ver.ChipFwAndPhyVersion.PhyVersion[3]);

    // Set connection policy to Auto + SmartConfig
    //      (Device's default connection policy)
    lRetVal = sl_WlanPolicySet(SL_POLICY_CONNECTION,
                                SL_CONNECTION_POLICY(1, 0, 0, 0, 1), NULL, 0);
    ASSERT_ON_ERROR(lRetVal);

    // Remove all profiles
    lRetVal = sl_WlanProfileDel(0xFF);
    ASSERT_ON_ERROR(lRetVal);

    //
    // Device in station-mode. Disconnect previous connection if any
    // The function returns 0 if 'Disconnected done', negative number if already
    // disconnected Wait for 'disconnection' event if 0 is returned, Ignore
    // other return-codes
    //
    lRetVal = sl_WlanDisconnect();
    if(0 == lRetVal) {
        // Wait
        while(IS_CONNECTED(g_ulStatus)) {
#ifndef SL_PLATFORM_MULTI_THREADED
              _SlNonOsMainLoopTask();
#endif
        }
    }

    // Enable DHCP client
    lRetVal = sl_NetCfgSet(SL_IPV4_STA_P2P_CL_DHCP_ENABLE,1,1,&ucVal);
    ASSERT_ON_ERROR(lRetVal);

    // Disable scan
    ucConfigOpt = SL_SCAN_POLICY(0);
    lRetVal = sl_WlanPolicySet(SL_POLICY_SCAN , ucConfigOpt, NULL, 0);
    ASSERT_ON_ERROR(lRetVal);

    // Set Tx power level for station mode
    // Number between 0-15, as dB offset from max power - 0 will set max power
    ucPower = 0;
    lRetVal = sl_WlanSet(SL_WLAN_CFG_GENERAL_PARAM_ID,
            WLAN_GENERAL_PARAM_OPT_STA_TX_POWER, 1, (unsigned char *)&ucPower);
    ASSERT_ON_ERROR(lRetVal);

    // Set PM policy to normal
    lRetVal = sl_WlanPolicySet(SL_POLICY_PM , SL_NORMAL_POLICY, NULL, 0);
    ASSERT_ON_ERROR(lRetVal);

    // Unregister mDNS services
    lRetVal = sl_NetAppMDNSUnRegisterService(0, 0);
    ASSERT_ON_ERROR(lRetVal);

    // Remove  all 64 filters (8*8)
    memset(RxFilterIdMask.FilterIdMask, 0xFF, 8);
    lRetVal = sl_WlanRxFilterSet(SL_REMOVE_RX_FILTER, (_u8 *)&RxFilterIdMask,
                       sizeof(_WlanRxFilterOperationCommandBuff_t));
    ASSERT_ON_ERROR(lRetVal);

    lRetVal = sl_Stop(SL_STOP_TIMEOUT);
    ASSERT_ON_ERROR(lRetVal);

    InitializeAppVariables();

    return lRetVal; // Success
}

//*****************************************************************************
//
//! Board Initialization & Configuration
//!
//! \param  None
```

```
//!
//! \return None
//
//*****************************************************************************
static void BoardInit(void) {
/* In case of TI-RTOS vector table is initialize by OS itself */
#ifndef USE_TIRTOS
  //
  // Set vector table base
  //
#if defined(ccs)
    MAP_IntVTableBaseSet((unsigned long)&g_pfnVectors[0]);
#endif
#if defined(ewarm)
    MAP_IntVTableBaseSet((unsigned long)&__vector_table);
#endif
#endif
    //
    // Enable Processor
    //
    MAP_IntMasterEnable();
    MAP_IntEnable(FAULT_SYSTICK);

    PRCMCC3200MCUInit();
}


//****************************************************************************
//
//! \brief Connecting to a WLAN Accesspoint
//!
//!  This function connects to the required AP (SSID_NAME) with Security
//!  parameters specified in te form of macros at the top of this file
//!
//! \param  None
//!
//! \return  0 on success else error code
//!
//! \warning    If the WLAN connection fails or we don't aquire an IP
//!            address, It will be stuck in this function forever.
//
//****************************************************************************
static long WlanConnect() {
    SlSecParams_t secParams = {0};
    long lRetVal = 0;

    secParams.Key = SECURITY_KEY;
    secParams.KeyLen = strlen(SECURITY_KEY);
    secParams.Type = SECURITY_TYPE;

    UART_PRINT("Attempting connection to access point: ");
    UART_PRINT(SSID_NAME);
    UART_PRINT("... ...");
    lRetVal = sl_WlanConnect(SSID_NAME, strlen(SSID_NAME), 0, &secParams, 0);
    ASSERT_ON_ERROR(lRetVal);

    UART_PRINT(" Connected!!!\n\r");
    // Wait for WLAN Event
    while((!IS_CONNECTED(g_ulStatus)) || (!IS_IP_ACQUIRED(g_ulStatus))) {
        // Toggle LEDs to Indicate Connection Progress
        _SlNonOsMainLoopTask();
        GPIO_IF_LedOff(MCU_IP_ALLOC_IND);
        MAP_UtilsDelay(800000);
        _SlNonOsMainLoopTask();
        GPIO_IF_LedOn(MCU_IP_ALLOC_IND);
        MAP_UtilsDelay(800000);
    }

    return SUCCESS;

}

//*****************************************************************************
//
//! This function updates the date and time of CC3200.
//!
//! \param None
//!
//! \return
//!     0 for success, negative otherwise
//!
//*****************************************************************************

static int set_time() {
    long retVal;

    g_time.tm_day = DATE;
    g_time.tm_mon = MONTH;
    g_time.tm_year = YEAR;
    g_time.tm_sec = HOUR;
    g_time.tm_hour = MINUTE;
    g_time.tm_min = SECOND;

    retVal = sl_DevSet(SL_DEVICE_GENERAL_CONFIGURATION,
                          SL_DEVICE_GENERAL_CONFIGURATION_DATE_TIME,
                          sizeof(SlDateTime),(unsigned char *)(&g_time));

    ASSERT_ON_ERROR(retVal);
    return SUCCESS;
}

long printErrConvenience(char * msg, long retVal) {
    UART_PRINT(msg);
    GPIO_IF_LedOn(MCU_RED_LED_GPIO);
    return retVal;
}

//*****************************************************************************
//
//! This function demonstrates how certificate can be used with SSL.
//! The procedure includes the following steps:
//! 1) connect to an open AP
//! 2) get the server name via a DNS request
//! 3) define all socket options and point to the CA certificate
//! 4) connect to the server via TCP
//!
//! \param None
//!
//! \return  0 on success else error code
//! \return  LED1 is turned solid in case of success
//!    LED2 is turned solid in case of failure
//!
//*****************************************************************************
static int tls_connect() {
    SlSockAddrIn_t    Addr;
    int    iAddrSize;
    unsigned char    ucMethod = SL_SO_SEC_METHOD_TLSV1_2;
    unsigned int uiIP,uiCipher = SL_SEC_MASK_TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA;
    long lRetVal = -1;
    int iSockID;

    lRetVal = sl_NetAppDnsGetHostByName(g_Host, strlen((const char *)g_Host),
                                    (unsigned long*)&uiIP, SL_AF_INET);

    if(lRetVal < 0) {
        return printErrConvenience("Device couldn't retrieve the host name \n\r", lRetVal);
    }

    Addr.sin_family = SL_AF_INET;
    Addr.sin_port = sl_Htons(GOOGLE_DST_PORT);
    Addr.sin_addr.s_addr = sl_Htonl(uiIP);
    iAddrSize = sizeof(SlSockAddrIn_t);
    //
    // opens a secure socket
    //
    iSockID = sl_Socket(SL_AF_INET,SL_SOCK_STREAM, SL_SEC_SOCKET);
    if( iSockID < 0 ) {
        return printErrConvenience("Device unable to create secure socket \n\r", lRetVal);
    }

    //
    // configure the socket as TLS1.2
    //
    lRetVal = sl_SetSockOpt(iSockID, SL_SOL_SOCKET, SL_SO_SECMETHOD, &ucMethod,\
                               sizeof(ucMethod));
    if(lRetVal < 0) {
        return printErrConvenience("Device couldn't set socket options \n\r", lRetVal);
    }
    //
    //configure the socket as ECDHE RSA WITH AES256 CBC SHA
    //
    lRetVal = sl_SetSockOpt(iSockID, SL_SOL_SOCKET, SL_SO_SECURE_MASK, &uiCipher,\
                           sizeof(uiCipher));
    if(lRetVal < 0) {
        return printErrConvenience("Device couldn't set socket options \n\r", lRetVal);
    }

    //
    //configure the socket with CA certificate - for server verification
    //
    lRetVal = sl_SetSockOpt(iSockID, SL_SOL_SOCKET, \
                           SL_SO_SECURE_FILES_CA_FILE_NAME, \
                           SL_SSL_CA_CERT, \
                           strlen(SL_SSL_CA_CERT));

    if(lRetVal < 0) {
        return printErrConvenience("Device couldn't set socket options \n\r", lRetVal);
    }

    //configure the socket with Client Certificate - for server verification
    //
    lRetVal = sl_SetSockOpt(iSockID, SL_SOL_SOCKET, \
                SL_SO_SECURE_FILES_CERTIFICATE_FILE_NAME, \
                                    SL_SSL_CLIENT, \
                           strlen(SL_SSL_CLIENT));

    if(lRetVal < 0) {
        return printErrConvenience("Device couldn't set socket options \n\r", lRetVal);
    }

    //configure the socket with Private Key - for server verification
    //
    lRetVal = sl_SetSockOpt(iSockID, SL_SOL_SOCKET, \
            SL_SO_SECURE_FILES_PRIVATE_KEY_FILE_NAME, \
            SL_SSL_PRIVATE, \
                           strlen(SL_SSL_PRIVATE));

    if(lRetVal < 0) {
        return printErrConvenience("Device couldn't set socket options \n\r", lRetVal);
    }


    /* connect to the peer device - Google server */
    lRetVal = sl_Connect(iSockID, ( SlSockAddr_t *)&Addr, iAddrSize);

    if(lRetVal < 0) {
        UART_PRINT("Device couldn't connect to server:");
        UART_PRINT(SERVER_NAME);
        UART_PRINT("\n\r");
        return printErrConvenience("Device couldn't connect to server \n\r", lRetVal);
    }
    else {
        UART_PRINT("Device has connected to the website:");
        UART_PRINT(SERVER_NAME);
        UART_PRINT("\n\r");
    }

    GPIO_IF_LedOff(MCU_RED_LED_GPIO);
    GPIO_IF_LedOn(MCU_GREEN_LED_GPIO);
    return iSockID;
}

int connectToAccessPoint() {
    long lRetVal = -1;
    GPIO_IF_LedConfigure(LED1|LED3);
    GPIO_IF_LedOff(MCU_RED_LED_GPIO);
    GPIO_IF_LedOff(MCU_GREEN_LED_GPIO);

    lRetVal = InitializeAppVariables();
    ASSERT_ON_ERROR(lRetVal);

    //
    // Following function configure the device to default state by cleaning
    // the persistent settings stored in NVMEM (viz. connection profiles &
    // policies, power policy etc)
    //
    // Applications may choose to skip this step if the developer is sure
    // that the device is in its default state at start of applicaton
    //
    // Note that all profiles and persistent settings that were done on the
    // device will be lost
    //
    lRetVal = ConfigureSimpleLinkToDefaultState();
    if(lRetVal < 0) {
      if (DEVICE_NOT_IN_STATION_MODE == lRetVal)
          UART_PRINT("Failed to configure the device in its default state \n\r");

      return lRetVal;
    }

    UART_PRINT("Device is configured in default state \n\r");

    CLR_STATUS_BIT_ALL(g_ulStatus);

    ///
    // Assumption is that the device is configured in station mode already
    // and it is in its default state
    //
    lRetVal = sl_Start(0, 0, 0);
    if (lRetVal < 0 || ROLE_STA != lRetVal) {
        UART_PRINT("Failed to start the device \n\r");
        return lRetVal;
    }

    UART_PRINT("Device started as STATION \n\r");

    //
    //Connecting to WLAN AP
    //
    lRetVal = WlanConnect();
    if(lRetVal < 0) {
        UART_PRINT("Failed to establish connection w/ an AP \n\r");
        GPIO_IF_LedOn(MCU_RED_LED_GPIO);
        return lRetVal;
    }

    UART_PRINT("Connection established w/ AP and IP is aquired \n\r");
    return 0;
}


//*****************************************************************************
```

```
//
// Main
//
//*****************************************************************************
void main() {
    // Initial board configuration.
    BoardInit();
    PinMuxConfig();

    // Terminal initialization.
    InitTerm();
    ClearTerm();

    // SPI initialization.
    MAP_PRCMPeripheralClkEnable(PRCM_GSPI,PRCM_RUN_MODE_CLK);
    MAP_PRCMPeripheralReset(PRCM_GSPI);
    MAP_SPIReset(GSPI_BASE);
    MAP_SPIConfigSetExpClk(GSPI_BASE,MAP_PRCMPeripheralClockGet(PRCM_GSPI),
                         SPI_IF_BIT_RATE,SPI_MODE_MASTER,SPI_SUB_MODE_0,
                         (SPI_SW_CTRL_CS |
                         SPI_4PIN_MODE |
                         SPI_TURBO_OFF |
                         SPI_CS_ACTIVELOW |
                         SPI_WL_8));
    MAP_SPIIntEnable(GSPI_BASE,SPI_INT_RX_FULL|SPI_INT_TX_EMPTY);
    MAP_SPIEnable(GSPI_BASE);

    // Adafruit (OLED) initialization.
    Adafruit_Init();

    // I2C initialization.
    I2C_IF_Open(I2C_MASTER_MODE_FST);

    // Clear the OLED.
    fillScreen(BLACK);

    // Register the IR remote interrupt handler.
    GPIOIntRegister(GPIOA1_BASE, IRRemoteInterruptHandler);
    // Configure falling edge interrupts for IR remote.
    GPIOIntTypeSet(GPIOA1_BASE, 0x1, GPIO_FALLING_EDGE);
    // Clear IR remote interrupts.
    unsigned long ulStatus = GPIOIntStatus (GPIOA1_BASE, false);
    GPIOIntClear(GPIOA1_BASE, ulStatus);

    // Initialize the bit-array.
    clearBitArray();

    // Initialize timer base addresses.
    g_ulBase = TIMERA0_BASE;

    // Configure the timers.
    Timer_IF_Init(PRCM_TIMERA0, g_ulBase, TIMER_CFG_PERIODIC_UP, TIMER_A, 255);

    // Setup the interrupt handlers for timer timeouts.
    Timer_IF_IntSetup(g_ulBase, TIMER_A, TimerBaseIntHandler);

    // Turn on the timers (time values in ms).
```

```
Timer_IF_Start(g_ulBase, TIMER_A, 50);

// Enable the IR remote interrupt.
GPIOIntEnable(GPIOA1_BASE, 0x1);

// Configure and initialize the UART for sending messages between boards.
UARTConfigSetExpClk(
    UARTA1_BASE,
    PRCMPeripheralClockGet(PRCM_UARTA1),
    UART_BAUD_RATE,
    (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE)
);
UARTEnable(UARTA1_BASE);


//Connect the CC3200 to the local access point
socketID = connectToAccessPoint();
//Set time so that encryption can be used
socketID = set_time();
if(socketID < 0) {
    UART_PRINT("Unable to set time in the device");
    LOOP_FOREVER();
}
//Connect to the website with TLS encryption
socketID = tls_connect();
if(socketID < 0) {
    ERR_PRINT(socketID);
}

// Initialize the board on AWS.
http_post(socketID, "turn", "W");
http_post(socketID, "lastmove", "xxxx");

// Initialize the messages.
message1.aliveCount = 0;
message2.aliveCount = 2;

// Reverse the player position for the black player.
if (thisPlayer == BLACK_PLAYER)
{
    cursorX = 7 - cursorX;
    cursorY = 7 - cursorY;
}

// Reset the local chess board state.
resetChessBoard();
// Redraw the board on the OLED.
redrawOLED(thisPlayer);

// The main program loop (never exits).
while(1)
{
    if (turn != thisPlayer)
    {
        // Wait a few seconds.
        int j = 0;
        for (j = 0; j < 25000000; j += 2)
```

```
            {
                j--;
            }


            char thisPlayerChar = 'W';
            if (thisPlayer == BLACK_PLAYER)
            {
                thisPlayerChar = 'B';
            }


            char rData[128];
            http_get(socketID, "turn", 4, rData, 1);
            http_get(socketID, "turn", 4, rData, 1);
            UART_PRINT("gotten: >");
            UART_PRINT(rData);
            UART_PRINT("<\n\r");

            if (rData[0] == thisPlayerChar)
            {
                http_get(socketID, "lastmove", 8, rData, 4);

                parseLastMove(rData, &lastMoveSrcX, &lastMoveSrcY, &lastMoveDestX,
&lastMoveDestY);


                doMove(lastMoveSrcX, lastMoveSrcY, lastMoveDestX, lastMoveDestY,
chessBoardState);
                turn = thisPlayer;

                if (inCheckMate(thisPlayer, chessBoardState))
                {
                    winner = getOtherPlayer(thisPlayer);

                    strcpy(message1.text, "You lose!");
                    message1.fgColor = WHITE;
                    message1.bgColor = BLUE;
                    message1.size = 2;
                    message1.xPos = 10;
                    message1.yPos = 40;
                    message1.aliveCount = 2;

                    strcpy(message2.text, "Press 0 to restart");
                    message2.fgColor = WHITE;
                    message2.bgColor = BLUE;
                    message2.size = 1;
                    message2.xPos = 10;
                    message2.yPos = 60;
                    message2.aliveCount = 2;
                }
                else
                {
                    strcpy(message1.text, "Your turn");
                    message1.fgColor = WHITE;
                    message1.bgColor = BLUE;
                    message1.size = 1;
                    message1.xPos = 10;
                    message1.yPos = 10;
                    message1.aliveCount = 2;
                    message2.aliveCount = 0;
                }

                redrawOLED(thisPlayer);
            }
        }
    }
}


static int http_get(int iTLSSockID, char* getFieldName, int getFieldNameLength, char* rcvData,
int rcvDataLength){
    UART_PRINT("************* http_get\n\r");

    char acSendBuff[512];
    char acRecvbuff[1460];
    char cCLLength[200];
    char* pcBufHeaders;
    int lRetVal = 0;

    char data[512] = "\0";

    pcBufHeaders = acSendBuff;
    strcpy(pcBufHeaders, GETHEADER);
    pcBufHeaders += strlen(GETHEADER);
    strcpy(pcBufHeaders, HOSTHEADER);
    pcBufHeaders += strlen(HOSTHEADER);
    strcpy(pcBufHeaders, CHEADER);
    pcBufHeaders += strlen(CHEADER);
    strcpy(pcBufHeaders, "\r\n\r\n");

    int dataLength = strlen(data);

    strcpy(pcBufHeaders, CTHEADER);
    pcBufHeaders += strlen(CTHEADER);
    strcpy(pcBufHeaders, CLHEADER1);

    pcBufHeaders += strlen(CLHEADER1);
    sprintf(cCLLength, "%d", dataLength);

    strcpy(pcBufHeaders, cCLLength);
    pcBufHeaders += strlen(cCLLength);
    strcpy(pcBufHeaders, CLHEADER2);
    pcBufHeaders += strlen(CLHEADER2);

    strcpy(pcBufHeaders, data);
    pcBufHeaders += strlen(data);

    int testDataLength = strlen(pcBufHeaders);

    UART_PRINT(acSendBuff);


    //
```

```
// Send the packet to the server */
//
lRetVal = sl_Send(iTLSSockID, acSendBuff, strlen(acSendBuff), 0);
if(lRetVal < 0) {
    UART_PRINT("POST failed. Error Number: %i\n\r",lRetVal);
    sl_Close(iTLSSockID);
    GPIO_IF_LedOn(MCU_RED_LED_GPIO);
    return lRetVal;
}
lRetVal = sl_Recv(iTLSSockID, &acRecvbuff[0], sizeof(acRecvbuff), 0);
if(lRetVal < 0) {
    UART_PRINT("Received failed. Error Number: %i\n\r",lRetVal);
    //sl_Close(iSSLSockID);
    GPIO_IF_LedOn(MCU_RED_LED_GPIO);
       return lRetVal;
}
else {
    UART_PRINT("GET acRecvbuff:\n\r");
    acRecvbuff[lRetVal+1] = '\0';
    UART_PRINT(acRecvbuff);
    UART_PRINT("\n\r\n\r");
    UART_PRINT("GET acRecvbuffEND\n\r");
}

// Find the field data in the received buffer.
rcvData[0] = '\0';
int i = 0;
int j;
int brk = 0;
int foundAt = -1;
while (!brk)
{
    for (j = 0; j < getFieldNameLength; j++)
    {
        if (i + j >= lRetVal)
        {
            brk = 1;
            break;
        }

        if (getFieldName[j] != acRecvbuff[i + j])
        {
            break;
        }
        else if (j == getFieldNameLength - 1)
        {
            foundAt = i;
            brk = 1;
        }
    }

    i++;
}
if (foundAt < 0)
{
    return 0;
}
```

```
    // Here, foundAt holds the index where the token was found. Put the correct data into the
receiveData buffer.
    i = foundAt + 3 + getFieldNameLength;
    for (j = 0; j < rcvDataLength; j++)
    {
        if (i + j >= lRetVal)
        {
            rcvData[0] = '\0';
            return 0;
        }

        rcvData[j] = acRecvbuff[i + j];
    }
    rcvData[rcvDataLength] = '\0';


    UART_PRINT("\n\r\n\r\n\r\n\r\n\r\n\r\n\r");
    return 0;
}


// Sends the current message to the cell phone using HTTP POST.
static int http_post(int iTLSSockID, char field[], char value[]){
    UART_PRINT("************* http_post\n\r");

    char acSendBuff[512];
//    char acRecvbuff[1460];
    char cCLLength[200];
    char* pcBufHeaders;
    int lRetVal = 0;

    pcBufHeaders = acSendBuff;
    strcpy(pcBufHeaders, POSTHEADER);
    pcBufHeaders += strlen(POSTHEADER);
    strcpy(pcBufHeaders, HOSTHEADER);
    pcBufHeaders += strlen(HOSTHEADER);
    strcpy(pcBufHeaders, CHEADER);
    pcBufHeaders += strlen(CHEADER);
    strcpy(pcBufHeaders, "\r\n\r\n");


    int dataLength = strlen(DATA_PREFIX) + strlen(field) + strlen(DATA_MIDFIX) + strlen(value)
+ strlen(DATA_POSTFIX);

    strcpy(pcBufHeaders, CTHEADER);
    pcBufHeaders += strlen(CTHEADER);
    strcpy(pcBufHeaders, CLHEADER1);

    pcBufHeaders += strlen(CLHEADER1);
    sprintf(cCLLength, "%d", dataLength);

    strcpy(pcBufHeaders, cCLLength);
    pcBufHeaders += strlen(cCLLength);
    strcpy(pcBufHeaders, CLHEADER2);
    pcBufHeaders += strlen(CLHEADER2);
    strcpy(pcBufHeaders, DATA_PREFIX);
    pcBufHeaders += strlen(DATA_PREFIX);
    strcpy(pcBufHeaders, field);
    pcBufHeaders += strlen(field);
    strcpy(pcBufHeaders, DATA_MIDFIX);
    pcBufHeaders += strlen(DATA_MIDFIX);
    strcpy(pcBufHeaders, value);
    pcBufHeaders += strlen(value);
    strcpy(pcBufHeaders, DATA_POSTFIX);
    pcBufHeaders += strlen(DATA_POSTFIX);

    int testDataLength = strlen(pcBufHeaders);

    UART_PRINT(acSendBuff);


    //
    // Send the packet to the server
    //
    lRetVal = sl_Send(iTLSSockID, acSendBuff, strlen(acSendBuff), 0);
    if(lRetVal < 0) {
        UART_PRINT("POST failed. Error Number: %i\n\r",lRetVal);
        sl_Close(iTLSSockID);
        GPIO_IF_LedOn(MCU_RED_LED_GPIO);
        return lRetVal;
    }
    // Only sending data to the server. Do not worry about receiving a response.
    /*lRetVal = sl_Recv(iTLSSockID, &acRecvbuff[0], sizeof(acRecvbuff), 0);
    if(lRetVal < 0) {
        UART_PRINT("Received failed. Error Number: %i\n\r",lRetVal);
        //sl_Close(iSSLSockID);
        GPIO_IF_LedOn(MCU_RED_LED_GPIO);
           return lRetVal;
    }
    else {
        acRecvbuff[lRetVal+1] = '\0';
        UART_PRINT(acRecvbuff);
        UART_PRINT("\n\r\n\r");
    }*/

    UART_PRINT("\n\r\n\r\n\r\n\r\n\r\n\r\n\r");
    return 0;
}
```

3) pinmux.c

```
#include "pinmux.h"
#include "hw_types.h"
#include "hw_memmap.h"
#include "hw_gpio.h"
#include "pin.h"
#include "rom.h"
#include "rom_map.h"
#include "gpio.h"
#include "prcm.h"
```

```
//*****************************************************************************
void
PinMuxConfig(void)
{
    //
    // Enable Peripheral Clocks
    //
    MAP_PRCMPeripheralClkEnable(PRCM_GPIOA1, PRCM_RUN_MODE_CLK);
    MAP_PRCMPeripheralClkEnable(PRCM_GPIOA3, PRCM_RUN_MODE_CLK);
    MAP_PRCMPeripheralClkEnable(PRCM_UARTA0, PRCM_RUN_MODE_CLK);

    //
    // Configure PIN_64 for GPIOOutput
    //
    MAP_PinTypeGPIO(PIN_64, PIN_MODE_0, false);
    MAP_GPIODirModeSet(GPIOA1_BASE, 0x2, GPIO_DIR_MODE_OUT);

    //
    // Configure PIN_02 for GPIOOutput
    //
    MAP_PinTypeGPIO(PIN_02, PIN_MODE_0, false);
    MAP_GPIODirModeSet(GPIOA1_BASE, 0x8, GPIO_DIR_MODE_OUT);


    //
    // Configure PIN_55 for UART0 UART0_TX
    //
    MAP_PinTypeUART(PIN_55, PIN_MODE_3);

    //
    // Configure PIN_57 for UART0 UART0_RX
    //
    MAP_PinTypeUART(PIN_57, PIN_MODE_3);


    //
    // Enable Peripheral Clocks
    //
    PRCMPeripheralClkEnable(PRCM_GPIOA0, PRCM_RUN_MODE_CLK);
    PRCMPeripheralClkEnable(PRCM_GPIOA3, PRCM_RUN_MODE_CLK);
    PRCMPeripheralClkEnable(PRCM_GSPI, PRCM_RUN_MODE_CLK);
    PRCMPeripheralClkEnable(PRCM_I2CA0, PRCM_RUN_MODE_CLK);

    PRCMPeripheralClkEnable(PRCM_GPIOA1, PRCM_RUN_MODE_CLK);
    PRCMPeripheralClkEnable(PRCM_GPIOA2, PRCM_RUN_MODE_CLK);

    PRCMPeripheralClkEnable(PRCM_UARTA1, PRCM_RUN_MODE_CLK);
    PRCMPeripheralClkEnable(PRCM_UARTA0, PRCM_RUN_MODE_CLK);

    //
    // Configure PIN_61 for GPIO Output
    //
    PinTypeGPIO(PIN_61, PIN_MODE_0, false);
    GPIODirModeSet(GPIOA0_BASE, 0x40, GPIO_DIR_MODE_OUT);

    //
    // Configure PIN_62 for GPIO Output
    //
    PinTypeGPIO(PIN_62, PIN_MODE_0, false);
    GPIODirModeSet(GPIOA0_BASE, 0x80, GPIO_DIR_MODE_OUT);

    //
    // Configure PIN_63 for GPIO Input
    //
    PinTypeGPIO(PIN_63, PIN_MODE_0, false);
    GPIODirModeSet(GPIOA1_BASE, 0x1, GPIO_DIR_MODE_IN);

    //
    // Configure PIN_18 for GPIO Output
    //
    PinTypeGPIO(PIN_18, PIN_MODE_0, false);
    GPIODirModeSet(GPIOA3_BASE, 0x10, GPIO_DIR_MODE_OUT);

    //
    // Configure PIN_08 for SPI0 GSPI_CS
    //
    PinTypeSPI(PIN_08, PIN_MODE_7);

    //
    // Configure PIN_05 for SPI0 GSPI_CLK
    //
    PinTypeSPI(PIN_05, PIN_MODE_7);

    //
    // Configure PIN_06 for SPI0 GSPI_MISO
    //
    PinTypeSPI(PIN_06, PIN_MODE_7);

    //
    // Configure PIN_07 for SPI0 GSPI_MOSI
    //
    PinTypeSPI(PIN_07, PIN_MODE_7);

    // SW
    //
    // Configure PIN_04 for GPIO Input
    //
    PinTypeGPIO(PIN_04, PIN_MODE_0, false);
    GPIODirModeSet(GPIOA1_BASE, 0x20, GPIO_DIR_MODE_IN);

    // SW
    //
    // Configure PIN_15 for GPIO Input
    //
    PinTypeGPIO(PIN_15, PIN_MODE_0, false);
    GPIODirModeSet(GPIOA2_BASE, 0x40, GPIO_DIR_MODE_IN);

    PinTypeUART(PIN_58, PIN_MODE_6); // TX A1
    PinTypeUART(PIN_59, PIN_MODE_6); // RX A1
}
```