Team Composition: Cara, Quinn, Lonnie

Introduction / Motivation for the current project

For our final project, we will build Jeopardy!. This idea was motivated by the project's potential for implementing software and hardware topics we've previously covered in class, while also providing challenges in possible growth areas. Hardware components include configuring circuits between an 8x8 LED matrix, LCDs, and button sensors that contestants use to buzz in. These pieces will be orchestrated by state machines to implement the game logic and control the LED and LCD display.

Design Requirements

Problem definition: We are building a Jeopardy! game because it seems like a fun project and is at an appropriate difficulty level. The components are as follows:

Physical setup:

- Four players: one host and three contestants.
- One joystick shield used from E29 Lab 4, which provides 4 buttons, a reset button, a joystick, and button that can be pressed by pressing the joystick into the shield. For our game, the 4 buttons and the joystick button are used and controlled by the host to select questions and accept/reject contestant answers.
- Three arcade buttons, one for each contestant, used to "buzz in" when answering questions.
- One of our visual components is a 8x8 LED RGB Matrix. The left and right columns are unused, leaving a 8x6 grid of LEDs. This is split into a 4x3 array of cells, where each cell is 2x2. The top 3x3 of cells are used for question selection, where the column corresponds to the topic and the row corresponds to the prize ($100, $200, or $300). The bottom 3 of cells each represent one of the three players.
- The same 16x2 LCD used in the PWM Audio Player lab, for displaying questions and how much prize money each contestant has won or lost depending on the game state.

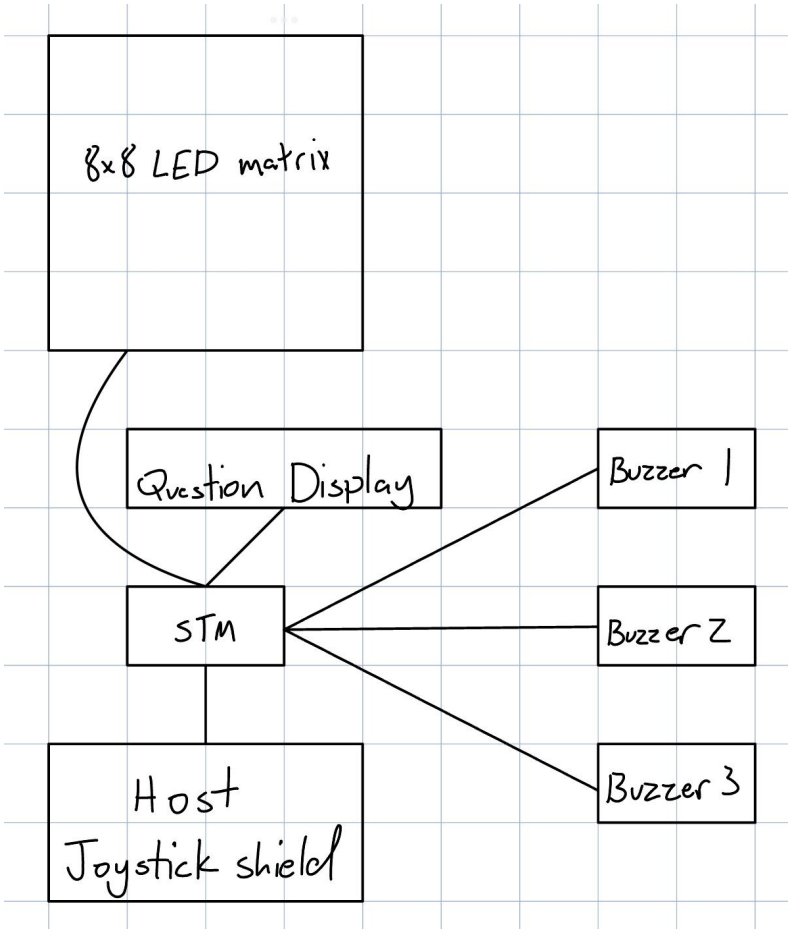
Gameplay:

1. The game is a big state machine, and powering on will start at the question selection state with 9 preprogrammed questions. During this state, the buzzers do nothing and the host can navigate around the 4x3 array of cells using the 4 buttons on the shield, where each is a 2x2 of LEDs on the matrix. The host's "cursor" is represented by lighting up the current cell yellow.
   If the host is hovering over a question (white for unanswered, black for answered), the LCD displays the topic and the prize amount for that question. If they hover over one of the bottom three cells (colored red, green, and blue), the score of the player with that color button will be displayed on the LCD. Note that the score can be negative if the player answers too many questions incorrectly.
2. When the host selects an unanswered question by pressing the joystick select button, the game enters a "countdown" state and the associated question is displayed on the LCD. After five seconds, the board flashes white and the first player to press their button

gets to answer the question. Note that during the countdown phase, any button presses will freeze the contestants buzzer for one second, so holding down the button while waiting is extremely disadvantageous.

3. Once a player buzzes in, the LED matrix flashes their color to signal to them that they were first. They can then give an answer, at which point the host either presses left (reject) or right (accept). If they accept, then that player gets points for the question and the game goes back to the question select state, except that question is now blocked. If the host rejects the answer, then they lose points equivalent to the prize of the question and another player who hasn't yet attempted that question can buzz in. If all three players miss the question, or if five seconds go by and nobody buzzes, then the game returns to the question select state and the question is blocked out. Note that if a contestant tries to hold down the buzzer right before the host rejects another contestant's answer, they will still have the one second early buzz penalty applied, putting them at a disadvantage.
4. Once all the questions are answered, the game is over and the player with the highest score wins! There's no special detection for this, but when all the questions are answered you can still navigate to the bottom and see who has the highest score.

Block Diagram



Hardware Implementation

The key components of our Jeopardy game include a breadboard, a STM32 Nucleo-64 microcontroller, a joystick shield, an 8x8 NeoPixel NeoMatrix LED display, and a 20x4 LCD to display questions to contestants. Our microcontroller, the STM32 Nucleo-64, is an integrated circuit that gathers inputs from the external environment, and executes outputs based on what it is programmed to do.

The joystick shield mounts to the microcontroller and transforms it into a simple controller. The joystick shield has five momentary push buttons (a two-axis joystick and four buttons) which are connected to the digital output pins 2-6, and when pressed they will pull the pin low. Vertical movement of the joystick produces a proportional analog voltage on analog pin 0, and horizontal movement of the joystick can be tracked on analog pin 1.

The 8x8 NeoMatrix LED display provides a visual display of the status of each question (answered or unanswered) and each participant. Each LED is split between one half RGB and the other half a white LED. They are controlled by 8-bit PWM per channel, so with 4 channels the LEDs produce a 32 bit color overall. Only one microcontroller pin is required to control all 64 LEDs. The 5V supplied by the microcontroller is a sufficient power supply for the LED panel.

The LCD can be controlled with six digital lines (any analog or digital pins), and the 20x4 LCD with the RGB background 3 PWM pins for the backlight. The 5V supplied by the microcontroller is sufficient for the power supply of these LCDs.

We were worried that the cords connecting the player buzzers to the breadboard could be easily ripped out of place during play. To make the position of the buttons more stationary, we cut holes in the side of a cardboard box and screwed the arcade buttons in. That way, the buttons could be slammed and sustain more rough usage, which adds to the “fun” component of our game.

Software: Hardware Interfacing Implementation

There are four pieces of hardware that our Nucleo interfaces with: the joystick shield, player buzzers, the 16x2 LCD, and the 8x8 LED matrix.

The joystick shield and the 16x2 LCD interfaces are implemented exactly the same as from the music player lab, so there’s nothing really special here. For the three player buttons, we drastically simplified our job by not using the I2C breakout connector and just connecting each button to a pin, which was then integrated into the software with a single BusIn.

For the 8x8 LED matrix, our interface code can be found in `headers/Matrix.h`. It consists of two types: a `NeoColor` struct, which is essentially just an RGB triple, and a `Matrix` class, which contains a `gpio_t` and an 8x8 buffer of `NeoColor`s. The idea is that we have helper functions like `fill` and `fill_rect` which can write colors onto the internal buffer, and then we have a `flush` method that flushes the contents of the buffer to the matrix by communicating via the protocol we reverse engineered.

The details of the protocol are as follows. When no information is being set, the GPIO pin idles at 0. When we want to draw to the board, we need to send over every bit of information all at once. The order is: for each row, for each column, for each uint8_t of (blue, red, green in that order) of the NeoColor at that row and column, for each bit in the uint8_t starting at the MSB and descending to the LSB: if it’s 0, do GPIO=1, NOP * 10, GPIO=0, NOP * 20; if it’s 1, do GPIO=1, NOP * 86, GPIO=0, NOP * 3. Our implementation is slightly obfuscated from this and

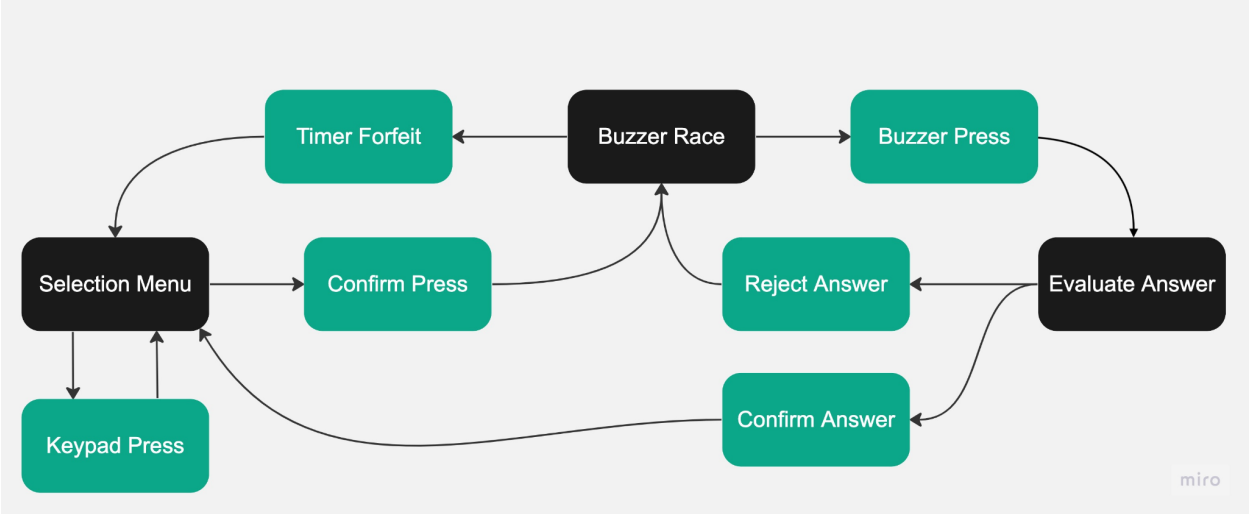
does some pointer casting that assumes memory layouts. This is because the implementation we reverse engineered did a similar thing and I didn't want to introduce any abstractions that might mess up the timings.

Software: Game Logic Implementation

Our state machine utilizes states for selecting a question, waiting for a player button press after a question has been chosen, and awarding points. When selecting a question, the program "hovers" an array of questions and colors the LEDs that correspond to the hovered question. It keeps track of variables such as which questions have already been answered and are out of play and individual player scores (prize winnings) as well.

When another button press confirms a question selection, the program moves into a waiting state. During the waiting state, the program starts with a short "dead" time interval and deducts points from scores if buttons are pressed. Button presses also activate a "timeout" refractory period if they are pressed during this dead time. After the dead time interval, the LEDs indicate that the next button press will activate the "answering" state and another timer begins. This new timer will move the program back into the question selection state if it is allowed to end.

The answering state moves the program back to the question selection state upon a certain button press or back to the waiting state if a different button is pressed. Points are awarded to the players' running totals accordingly.



Growth Areas

The two focus areas of this project were to use new hardware via the LED matrix and making a class / module in C++. For the LED matrix, we figured out how to interface with it by going through its Arduino interface, which was a new challenge for us. For software, our project involved a lot of state and controlling of LEDs, and this logic encapsulated abstractions that C++ offers.

Reflection

The final model of our Jeopardy! game was largely what we had envisioned it to be. The end user gameplay experience was successful, wherein key components such as the 8x8

NeoMatrix LED display functioned very well as a game board, the buzzers were responsive to user input, the host could control the game using the joystick shield, and game logic yielded smooth gameplay. Two major challenges we encountered were incorporating the LED arcade button I2C breakout and the 8x8 LED matrix into our design.

We ordered an arcade button “breakout” that plugged into the buttons and was supposed to make it easy to program the buttons. The breakout would be able to use built-in libraries to easily send signals from the buttons to the program. We encountered significant library compatibility issues. The breakout was designed to work with an Arduino board and the Arduino IDE. We were attempting to instead use it with a Nucleo board and our coding environment was MBed. We eventually realized that the devices we were trying to program were just buttons, and we knew how to work with buttons already from previous lab assignments in E29. Instead of plugging the buttons into the breakout, we soldered breadboard wires to them and connected them to the circuit and the pins on our Nucleo board, which we got working almost immediately.

The important takeaway from our experience struggling with the breakout is that even though the breakout was designed to make our job easier, sometimes it is better to just do it yourself. We understood what components of the button system we needed and how to implement them, and once we gave up on the breakout we managed to get the hardware working almost immediately.

The second challenge was getting the 8x8 LED matrix to work. We found a Nucleo library for it, except it was for a slightly different board model and wasn’t compatible with our Nucleo. Fortunately, the library only consisted of three short files: a header, an impl, and an assembly file. We were able to reverse engineer the communication protocol and write our own interface using Mbed’s GPIO struct and some inline assembly.

Future work on this project can include implementing new hardware components to the game. Adding a PWM audio player to play Jeopardy! music would add another dimension to enhance the experience, while also building off of and hopefully improving what we have done in a previous E29 lab. Additionally, because of time constraints, we used the 16x2 LCD in our project. With future work, we hope to use a larger display, the 20x4 LCD, to present questions in our game. Translating additional code from Arduino libraries to Mbed OS for the 20x4 LCD was not something we were able to do given the scope of this project, but a future continuation of this project can certainly work to include that.

Appendix A: Main.cpp code

```
/* mbed Microcontroller Library
 * Copyright (c) 2019 ARM Limited
 * SPDX-License-Identifier: Apache-2.0
 */

#include "mbed.h"
#include "headers/NHD_0216HZ.h"
#include "headers/Matrix.h"

// The amount of time to wait for people to read the question
#define READING_TIME_SECONDS 5

// The amount of time to wait for a buzz before everyone forfeits
#define AWAITING_TIME_SECONDS 5

// Enable for debug printing
#define DEBUG true

using namespace std::chrono;

BusIn buzzers(ARDUINO_UNO_D9, ARDUINO_UNO_D8, ARDUINO_UNO_D7);
DigitalIn left(D6);
DigitalIn right(D3);
DigitalIn up(D4);
DigitalIn down(D5);
DigitalIn joystick(D2);
NHD_0216HZ lcd(SPI_CS, SPI_MOSI, SPI_SCK);
Matrix matrix(D12);

bool host_pressed_left() {
    return left == 0;
}

bool host_pressed_right() {
    return right == 0;
}

bool host_pressed_up() {
```

```
    return up == 0;
}

bool host_pressed_down() {
    return down == 0;
}

bool host_pressed_joystick() {
    return joystick == 0;
}

#define Player1 (0b001)
#define Player2 (0b010)
#define Player3 (0b100)

int poll_buzzers() {
    return buzzers.read() & buzzers.mask();
}

enum State {
    Init,
    Selection,
    NavLeft,
    NavRight,
    NavUp,
    NavDown,
    // Redraw the LED board after moving the "cursor"
    ReturnToSelection,
    HostReading,
    AwaitingBuzz,
    Answer,
};

struct Msg {
    // Potentially not null terminated
    char line1[16];
    char line2[16];
```

```
    Msg(const char l1[], const char l2[]) {
        bool finished1 = false;
        bool finished2 = false;
        for (uint8_t i = 0; i < 16; ++i) {
            finished1 = finished1 || (l1[i] == '\0');
            finished2 = finished2 || (l2[i] == '\0');
            line1[i] = finished1 ? '\0' : l1[i];
            line2[i] = finished2 ? '\0' : l2[i];
        }
    }

    Msg(const Msg&) = default;

    void to_global_lcd() const {
        lcd.clr_lcd();
        lcd.set_cursor(0, 0);
        lcd.printf("%.16s", line1);
        lcd.set_cursor(0, 1);
        lcd.printf("%.16s", line2);
    }
};

struct Question {
    bool already_answered;
    Msg question;
    Msg answer;

    Question(Msg question, Msg answer):
        already_answered(false),
        question(question),
        answer(answer) {}

    Question(const Question&) = default;
};

struct Player {
    Timer since_last_click;
    int score;
```

```
    Player(): score(0) {
        since_last_click.start();
    }

    // returns true and resets the timer if it's been 1 second since
the most
    // recent click, otherwise returns false and doesn't reset the
timer.
    auto try_click() -> bool {
        if
(duration_cast<seconds>(since_last_click.elapsed_time()).count() >= 1)
{
            since_last_click.reset();
            return true;
        }

        return false;
    }
};

Player players[3];

/// A title and three questions
struct Topic {
    const char* title;
    Question questions[3] = {
        Question(Msg("", ""), Msg("", "")),
        Question(Msg("", ""), Msg("", "")),
        Question(Msg("", ""), Msg("", ""))
    };

    Topic(const char title[], Question q1, Question q2, Question q3) {
        this->title = title;
        questions[0] = q1;
        questions[1] = q2;
        questions[2] = q3;
    }
```

```
    Topic(const Topic&) = default;
};

struct DisplayState {
    Topic topics[3];

    DisplayState(Topic t1, Topic t2, Topic t3) : topics{t1, t2, t3} {}

    // Draws the colors of each question on the global matrix.
    // Black if the question has been answered, white otherwise.
    void write_on_global_matrix() {
        for (uint8_t row = 0; row < 3; ++row) {
            for (uint8_t col = 0; col < 3; ++col) {
                NeoColor color = is_unanswered(row, col)
                    ? NeoColor(20, 20, 20)
                    : NeoColor(0, 0, 0);

                matrix.fill_rect(
                    color,
                    (col * 2) + 1, // x
                    (row * 2), // y
                    2, // width
                    2 // height
                );
            }
        }
    }

    // Writes the topic and the $$$ on the LCD.
    // row must be in (0, 1, 2, 3) and col in (0, 1, 2)
    void write_nav_prompt_on_lcd(uint8_t row, uint8_t col) {
        lcd.clr_lcd();
        if (row < 3) {
            // Write the topic and prize amount
            lcd.set_cursor(0, 0);
            lcd.printf("%.16s", topics[col].title);
            lcd.set_cursor(0, 1);
```

```
            lcd.printf("$%d", 100 * (row + 1));
            if (DEBUG) { printf("%s\n$%d\n", topics[col].title, 100 *
(row + 1)); }
        } else if (row == 3) {
            // Write the player score given their col
            lcd.set_cursor(0, 0);
            lcd.printf("Player %d", col + 1);
            lcd.set_cursor(0, 1);
            lcd.printf("$%d", players[col].score);
            if (DEBUG) { printf("Player %d\n$%d\n", col + 1,
players[col].score); }

        } else {
            if (DEBUG) { printf("ERROR: ENTERED INVALID STATE\n"); }
        }
    }

    bool is_unanswered(uint8_t row, uint8_t col) {
        return !topics[col].questions[row].already_answered;
    }
};


#define BRIGHTNESS 20

void setup() {
    // Buzzers
    buzzers.mode(PullUp);

    // Host buttons
    left.mode(PullUp);
    right.mode(PullUp);
    up.mode(PullUp);
    down.mode(PullUp);
    joystick.mode(PullUp);

    // LCD
    lcd.init_lcd();
```

```
    lcd.clr_lcd();
}

int main() {
    setup();
    // Game state
    State state = Init;

    DisplayState display(
        Topic("Professors",
            Question(////////////////
                Msg("Carr's favorite",
                    "candy?"),
                Msg("Peanut M&Ms", "")
            ),
            Question(////////////////
                Msg("What does",
                    "Molter collect?"),
                Msg("Potato mashers", "")
            ),
            Question(////////////////
                Msg("What online game",
                    "Delano top 1%?"),
                Msg("Hearthstone", "")
            )
        ),
               ////////////////
        Topic("Guess lyrics",
            Question(////////////////
                Msg("I hopped off the ",
                    "plane at LAX"),
                Msg("Party in the",
                    "U.S.A.")
            ),
            Question(////////////////
                Msg("A tornado flew",
                    "around my room"),
                Msg("Thinkin Bout You", "")
```

```
        ),
        Question(///////////////
            Msg("moms spaghetti", ""),
            Msg("Lose Yourself", "")
        )
    ),
    Topic("Rhyme Time",
        Question(///////////////
            Msg("An overfed ",
                "feline"),
            Msg("Fat Cat", "")
        ),
        Question(///////////////
            Msg("An artificial ",
                "small horse"),
            Msg("Phony Pony", "")
        ),
        Question(///////////////
            Msg("Rose colored ",
                "water basin"),
            Msg("Pink Sink", "")
        )
    )
);

Timer reading_timer;
Timer awaiting_timer;
uint8_t who_buzzed_idx;

// Using (row, col) notation:
// (0, 0) | (0, 1) | (0, 2)
// -------+--------+-------
// (1, 0) | (1, 1) | (1, 2)
// -------+--------+-------
// (2, 0) | (2, 1) | (2, 2)
// -------+--------+-------
// (3, 0) | (3, 1) | (3, 2)
uint8_t row = 0;
```

```
    uint8_t col = 0;

    uint8_t block_from_buzzing_flags = 0b000;

    while (true) {
        switch (state) {
            case Init: {
                if (DEBUG) { printf("Init\n"); }
                state = ReturnToSelection;
                break;
            } // end Init
            case Selection: {
                if (host_pressed_left()) {
                    state = NavLeft;
                } else if (host_pressed_right()) {
                    state = NavRight;
                } else if (host_pressed_up()) {
                    state = NavUp;
                } else if (host_pressed_down()) {
                    state = NavDown;
                } else if (host_pressed_joystick() && row < 3 &&
display.is_unanswered(row, col)) {
                    reading_timer.reset();
                    reading_timer.start();
                    if (DEBUG) { printf("Selected question: topic: %s,
row: %d\n", display.topics[col].title, row); }

                    // Make the board black
                    matrix.fill(NeoColor());
                    matrix.flush();

                    Msg& q =
display.topics[col].questions[row].question;
                    if (DEBUG) { printf("%.16s %.16s\n", q.line1,
q.line2); }
                    q.to_global_lcd();

                    state = HostReading;
```

```
    }
    break;
} // end SelectQuestion
case NavLeft: {
    if (col > 0) {
        // moving left a question
        col -= 1;
    }
    state = ReturnToSelection;
    break;
} // end NavLeft
case NavRight: {
    if (col < 2) {
        // moving right a question
        col += 1;
    }
    state = ReturnToSelection;
    break;
} // end NavRight
case NavUp: {
    if (row > 0) {
        // moving up a question
        row -= 1;
    }
    state = ReturnToSelection;
    break;
} // end NavUp
case NavDown: {
    if (row < 3) {
        // moving down a question
        row += 1;
    }
    state = ReturnToSelection;
    break;
} // end NavDown
case ReturnToSelection: {
    if (DEBUG) { printf("row: %d, col: %d\n", row, col); }
```

```
                // Write stuff back to LCD
                display.write_nav_prompt_on_lcd(row, col);

                // Set everything to white
                matrix.fill_rect(NeoColor(20, 20, 20), 0, 0, 8, 8);

                // Fill in the question part of the board
                display.write_on_global_matrix();

                // Draw the "cursor"
                uint8_t cx = (col * 2) + 1;
                uint8_t cy = (row * 2);
                matrix.fill_rect(NeoColor(20, 20, 0), cx, cy, 2, 2);

                // Draw the three player colors
                uint8_t b[3] = { 10, 10, 10 };
                if (row == 3) {
                    b[col] = 30;
                }

                matrix.fill_rect(NeoColor(b[0], 0, 0), 1, 6, 2, 2);
                matrix.fill_rect(NeoColor(0, b[1], 0), 3, 6, 2, 2);
                matrix.fill_rect(NeoColor(0, 0, b[2]), 5, 6, 2, 2);

                // Flush
                matrix.flush();

                state = Selection;
                ThisThread::sleep_for(200ms);
                break;
            } // end ReturnToSelection
            case HostReading: {
                // Check if it's been the proper amount of time
                if
(duration_cast<seconds>(reading_timer.elapsed_time()).count() >=
READING_TIME_SECONDS) {
                    reading_timer.stop();
                    matrix.fill(NeoColor(20, 20, 20));
```

```
                    matrix.flush();
                    Msg& answer =
display.topics[col].questions[row].answer;
                    printf("Answer: %.16s %.16s\n", answer.line1,
answer.line2);

                    awaiting_timer.reset();
                    awaiting_timer.start();
                    state = AwaitingBuzz;
                }

                // but also prepare to penalize someone...

                uint8_t flags = poll_buzzers();
                if (flags & Player1) {
                    players[0].since_last_click.reset();
                }
                if (flags & Player2) {
                    players[1].since_last_click.reset();
                }
                if (flags & Player3) {
                    players[2].since_last_click.reset();
                }
                break;
            } // end HostReading
            case AwaitingBuzz: {
                if (block_from_buzzing_flags == 0b111
                    ||
duration_cast<seconds>(awaiting_timer.elapsed_time()).count() >=
AWAITING_TIME_SECONDS
                ) {
                    // Everyone failed to answer OR nobody wanted to
buzz in

display.topics[col].questions[row].already_answered = true;
                    block_from_buzzing_flags = 0b000;
                    state = ReturnToSelection;
                }
```

```
                uint8_t flags = poll_buzzers();
                if (flags & Player1 && players[0].try_click() &&
((block_from_buzzing_flags & Player1) == 0)) {
                    who_buzzed_idx = 0;
                    matrix.fill(NeoColor(10, 0, 0));
                    matrix.flush();
                    state = Answer;
                }
                if (flags & Player2 && players[1].try_click() &&
((block_from_buzzing_flags & Player2) == 0)) {
                    who_buzzed_idx = 1;
                    matrix.fill(NeoColor(0, 10, 0));
                    matrix.flush();
                    state = Answer;
                }
                if (flags & Player3 && players[2].try_click() &&
((block_from_buzzing_flags & Player3) == 0)) {
                    who_buzzed_idx = 2;
                    matrix.fill(NeoColor(0, 0, 10));
                    matrix.flush();
                    state = Answer;
                }
                break;
            } // end AwaitingBuzz
            case Answer: {
                // wait until the host either confirms or denies
                int points = 100 * (row + 1);
                if (host_pressed_right()) { // accept
                    players[who_buzzed_idx].score += points;


display.topics[col].questions[row].already_answered = true;
                    block_from_buzzing_flags = 0b000;
                    state = ReturnToSelection;
                } else if (host_pressed_left()) { // reject
                    players[who_buzzed_idx].score -= points;
                    block_from_buzzing_flags |= (1 << who_buzzed_idx);
                    matrix.fill(NeoColor(20, 20, 20));
```

```
                    matrix.flush();
                    awaiting_timer.reset();
                    awaiting_timer.start();
                    state = AwaitingBuzz;
                }

                // reset player timers if they buzz too early
                // e.g. they know the answer was wrong and try to spam
click
                uint8_t flags = poll_buzzers();
                if (flags & Player1) {
                    players[0].since_last_click.reset();
                }
                if (flags & Player2) {
                    players[1].since_last_click.reset();
                }
                if (flags & Player3) {
                    players[2].since_last_click.reset();
                }
                break;
            } // end Answer
        } // end switch
    } // end event loop
} // end main
```

Appendix B: Header file for matrix

```
#ifndef HEADERS_MATRIX
#define HEADERS_MATRIX

#include "mbed.h"

#define NOP_10
__asm__("NOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n
\tNOP")
#define NOP_20
__asm__("NOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n
\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n
\tNOP")
```

```
#define NOP_86
__asm__("NOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n
\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n
\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n
\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n
\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n
\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n
\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n
\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n
\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP\n\tNOP")
#define NOP_3 __asm__("NOP\n\tNOP\n\tNOP")


typedef struct _NeoColor {
    uint8_t blue;
    uint8_t red;
    uint8_t green;

    _NeoColor(uint8_t red = 0, uint8_t green = 0, uint8_t blue = 0) :
red(red), green(green), blue(blue) {}

    _NeoColor operator+(_NeoColor other) {
        return _NeoColor(red + other.red, green + other.green, blue +
other.blue);
    }

    _NeoColor operator*(uint8_t k) {
        return _NeoColor(red * k, green * k, blue * k);
    }
} NeoColor;

class Matrix {
public:
    Matrix(PinName pin) {
        gpio_init_out(&gpio, pin);
        fill(20, 20, 20);
    }
```

```
    void flush() {
        for (uint8_t byte_loop = 64*3; byte_loop != 0; --byte_loop) {
            uint8_t byte_to_send = ((uint8_t*)pixels)[byte_loop-1];
            for (uint8_t bit_mask = 0x80; bit_mask != 0; bit_mask >>=
1) {
                if ((byte_to_send & bit_mask) == 0) {
                    // send 0
                    // Output a NeoPixel zero, composed of a short
                    // HIGH pulse and a long LOW pulse
                    gpio_write(&gpio, 1);
                    // NOP 10 times
                    NOP_10;
                    gpio_write(&gpio, 0);
                    // NOP 20 times
                    NOP_20;
                } else {
                    // send 1
                    gpio_write(&gpio, 1);
                    // NOP 86 times
                    NOP_86;

                    gpio_write(&gpio, 0);
                    // NOP 3 times
                    NOP_3;
                }
            }
        }
        gpio_write(&gpio, 0);
        // wait for the reset pulse
        wait_us(50);
        // ThisThread::sleep_for(1ms);
    }

    void fill(uint8_t green, uint8_t red, uint8_t blue) {
        for (int row = 0; row < 8; ++row) {
            for (int col = 0; col < 8; ++col) {
                pixels[row][col].green = green;
                pixels[row][col].red = red;
```

```
                pixels[row][col].blue = blue;
            }
        }
    }

    int fill_rect(NeoColor color, uint8_t x, uint8_t y, uint8_t width,
uint8_t height) {
        if (x + width > 8 || y + height > 8) { return 1; }
        // print them in row-major order
        for (uint8_t row = 0; row < height; ++row) {
            for (uint8_t col = 0; col < width; ++col) {
                pixels[y + row][x + col] = color;
            }
        }

        return 0;
    }

    void fill(NeoColor color) {
        for (uint8_t row = 0; row < 8; ++row) {
            for (uint8_t col = 0; col < 8; ++col) {
                pixels[row][col] = color;
            }
        }
    }


private:
    gpio_t gpio;
    NeoColor pixels[8][8];
};

#endif // HEADERS_MATRIX
```

Appendix C: Header file for Newhaven NHD0216HZ LCD

```
#ifndef NHD_0216HZ_H
#define NHD_0216HZ_H
```

```
#include "mbed.h"

//Define constants
#define ENABLE 0x08
#define DATA_MODE 0x04
#define COMMAND_MODE 0x00

#define LINE_LENGTH 0x40
#define TOT_LENGTH 0x80

class NHD_0216HZ{
      public:
            NHD_0216HZ(PinName SPI_CS, PinName SPI_MOSI, PinName
SPI_SCLK);

            //Function prototypes
            void shift_out(int data);
            void init_lcd(void);
            void write_4bit(int data, int mode);
            void write_cmd(int data);
            void write_data(char c);
           void printf(const char *format, ...);
            void set_cursor(int column, int row);
            void clr_lcd(void);

      private:
            DigitalOut _SPI_CS;
            DigitalOut _SPI_MOSI;
            DigitalOut _SPI_SCLK;
};
#endif

// ********************************ARM University Program Copyright (c)
ARM Ltd 2014*************************************
```