1 4 9 5

UNIVERSITY OF ABERDEEN

School of Engineering

Academic Year: 2020/21
Course: *EE3579 EE Design*
Assignment Title: **LAB Activity 1**

Student details

Perform the tree steps below:

(1) In the table below: Replace text within the # # symbols with your data; do **not delete** the # # symbols.

First Name:	[# *Krisjanis* #]
Family Name:	[# *Visnevskis* #]
Student ID:	[# 5_1_7_6_8_1_9_7 #]
Digit:	[1st_2nd_3rd_4th_5th_6th_7th_8th]

(2) Update the document header: Double click on the **page header**; Press "**CTR** + **A**"; Press **F9** (alternatively, right click "Update Field").

(3) Read the Plagiarism form on page 2; then update all fields**:** Double click on the page; Press "**CTR** + **A**"; Press **F9** (alternatively, right click "Update Field")**.**

Plagiarism Awareness Declaration Form

Date received: 15 March 2021

SCHOOL OF ENGINEERING

PLAGIARISM AWARENESS DECLARATION

Course Code …… EE3579 ……
SURNAME/FAMILY NAME: ……# *Visnevskis* #……..

FIRST NAME: ……# *Krisjanis* #……..
ID Number: ……# 5_1_7_6_8_1_9_7 #…….

You MUST *read the statement on "Cheating" and definition of "Plagiarism" contained in the Code of Practice on Student Discipline, Appendix 5.15 of the Academic Quality Handbook at:*
www.abdn.ac.uk/registry/quality/appendices.shtml#section5

I confirm that I have read, understood and will abide by the University statement on cheating and plagiarism as provided in Academic Quality Handbook, and I have been made aware of how to correctly reference materials in all my submitted work, including all my Honours project reports and thesis. I have also read and understood the penalties where cheating and/or plagiarism are detected and proven as described in the University's Code of Practice on Student Discipline.

Signed:_____# *Krisjanis* # _# *Visnevskis* #____
Date:_____15 March 2021_____

EE3579: ACT 1; # *Krisjanis* # # *Visnevskis* #; # 5_1_7_6_8_1_9_7 #

Prerequisites:
See Guidelines_activity_1.pdf available via MyAberdeen for details on the system and the set tasks.

Report Preparation and Submission:
Each student writes a report describing the system and submits via MyAberdeen the **report** (as **PDF**) along with any library or sketch that has been written or modified for this system (one zip file).

Marking: The assignment can be solved to 4 degrees of complexity, leading to increasing marks, as follows:

BASIC:	**Section 1.**
LOW-INTERMEDIATE:	**(up to) Section 2.**
HIGH-INTERMEDIATE:	**(up to) Section 3.**
ADVANCED:	**(up to) Section 4.**

1 Basic

1.1 Overview of the system and its units

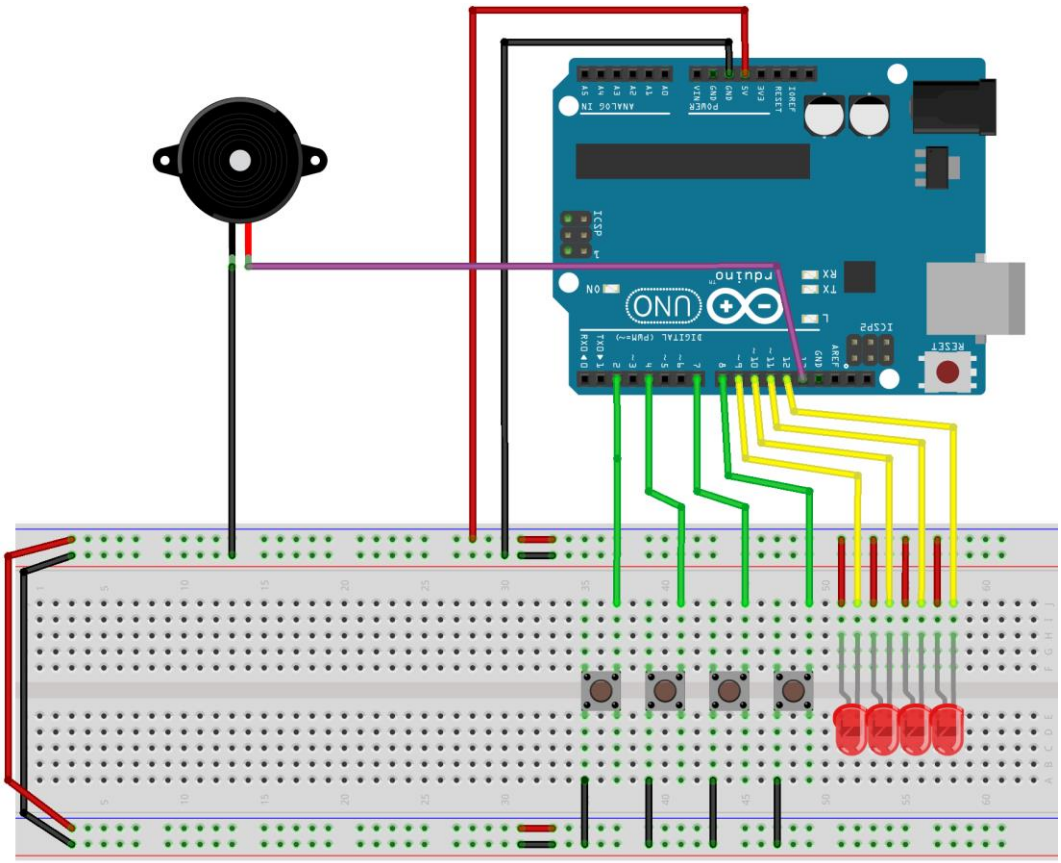
INSTRUCTIONS (remove TEXT IN RED before submission): Fill table below, depending on your student ID. Add rows as needed

System selections, based on student ID:

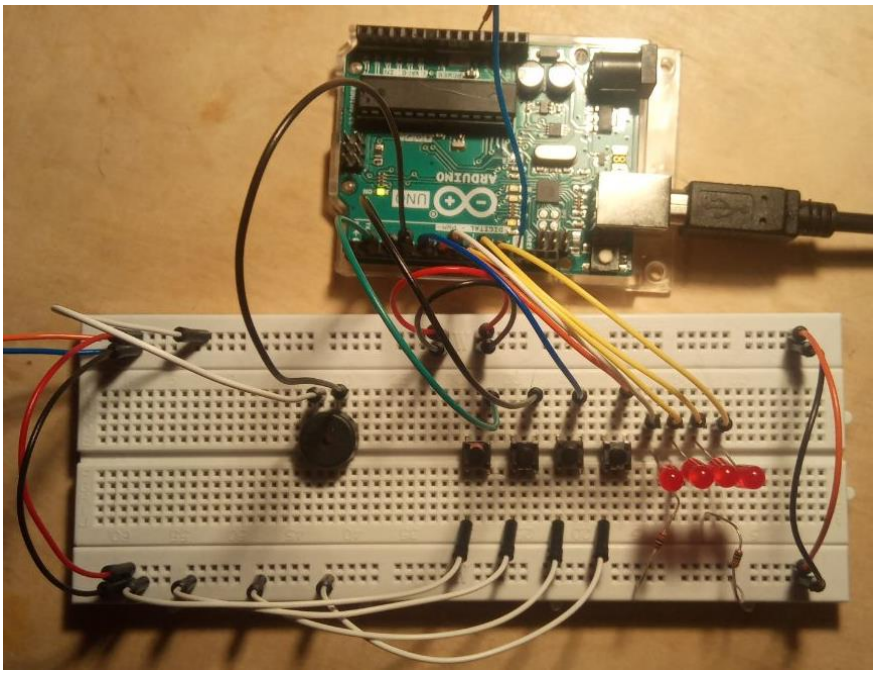
System Behaviour	**Relevant student ID (5^{th}, 6^{th}, 7^{th} or 8^{th})**	**Selection**
Cue Device	8^{th} digit is 7, odd	Buzzer with 4 distinct frequencies
Player input		4 pushbuttons, (up to 5 possible but I only have 4)
Input action feedback	7^{th} digit is 9, odd	Shorter feedback buzz than cue
Game and match score	6^{th} digit is 1, odd	Match is 6 games, with tie break
Game clock	5^{th} digit is 8, even	Clock expires if player response to any cue is longer than T_{resp}
Game outcome trigger	8^{th} digit is 7, range [5, 9]	Game ends at first player mistake, or clock expires or L correct responses.
Game/match outcome signal		Produce two buzzing sequences to signify win/loss of game and match
Match difficulty level	7^{th} digit is 9, range [5, 9]	Cues sequence speed increases, T_{resp} shirnks
Match difficulty level selection	7^{th} digit is 9, range [5, 9]	Warn player, provide selection feedback via Buzzer, wait and confirm (buzz).
Cue sequence generation	6^{th} digit is 1, range [0, 4]	Changes every game, pseudo random generation.

1.2 Hardware

The hardware consists of the Arduino Uno, a breadboard and leads, a buzzer for output, 4 pushbuttons for input and an optional 4 LEDs for output and resistors. The 4 LEDs make it easier to play the game for me as it is easier to memorise a sequence of visual cues than it is to memorise a sequence of aural cues. A diagram of the wiring is shown below:

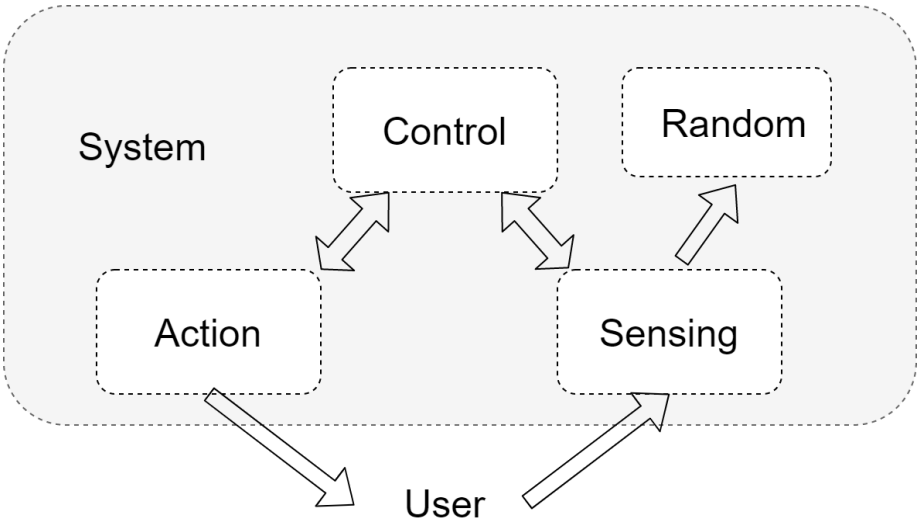


In reality I used only 2 resistors, 2 LEDs connected to each resistor. And some of the wiring was slightly different. The board layout was done in such a way as to not make it easy for user to press the buttons and also leave plenty of room for expansion if I had managed to get to advanced part of the assignment Pin connections do correspond to real life. A picture of the actual set up is shown below.



Software

The software consists of 5 units (classes): sensing, action, control, random and system. All of the units interact with system unit. Their interfaces shown in block diagram below:



Action unit

Task1_action.h class **buzzer_unit**

Main functionality:

- play the randomly generated sequence
- play feedback on user input
- play preset sequences (game win, game loss, match win, match loss, warning, countdown)

Main functions

- playCue() – takes cue and tone length as input, plays the required tone on the buzzer. This function is called to play feedback or called as part of playCueSequence() to play a cue sequence. A flag is used to signify if the input cue is feedback or not.
- playCueSequence() – takes array holding the sequence, tone length and cue length as inputs, calls playCue() in a loop to play the sequence

Additionally this unit includes the Beginner_LED library in order to display the cues as both buzzer output and as led lights. The buzzer output is produced using tone() function. To initialise the buzzer only a buzzer pin has to be specified.

```
#ifndef TASK1_ACTION_H
#define TASK1_ACTION_H
#include <Task1_sensing.h>
#include <Beginner_LED.h> //imcluded to visualise buzzer output
class buzzer_unit {
protected:
    bool init_flag;     //initialisation flag
    int buzzer_pin;     //pin for buzzer output
    const static int tones[5] = {392, 523, 660, 784, 100};  //tones for playing cues/feedback. not changeable by user.
    dig_LED led[4];     //only used to visualize tones
public:
    buzzer_unit() { ... }
    buzzer_unit(int pin_number) { ... }
    void setUpBuzzer(int pin_number) { ... }
    bool isInitialised() { return init_flag; }
    //next section is all the different tone sequences, they are set to specific lengths, not intended to be changable by user
    void countdown() { ... }
    void gameWin() { ... }
    void gameLoss() { ... }
    void matchWin() { ... }
    void matchLoss() { ... }
    void warning() { ... }
    void playCue(cmdEnum_push_button::cmdEnum button, bool isFeedback, int cue_length) { ... }
    void playCueSequence(cmdEnum_push_button::cmdEnum* sequence, int sequence_length, int cue_length) { ... }

};
```

Sensing unit

Task1_sensing.h class **button_input**

Main functionality

- Read user input

Main functions

- isButtonPressed() – returns true if any button press is detected at the time the function is called
- getPlayerInput() – returns an enumerator corresponding to an input device that was last pressed by the user

The function is initialised by passing an array of pin punmbers to it, the value for longpush detection and the number of pins to be used. The unit then declares and array of cmdEnum push buttons (from cmdEnumPushButton.h library). Assigning pin numbers and labels to the input devices. This unit also contains the waitForInput() function which halts the program until any user input is detected. Additionally the unit contains a getButtonsUsed() function which returns the number of input devices initialised. This value is used by the random unit to populate the random sequence.

```
constexpr cmdEnum_push_button::cmdEnum lables[5] = { cmdEnum_push_button::button1, cmdEnum_push_button::button2,
cmdEnum_push_button::button3, cmdEnum_push_button::button4, cmdEnum_push_button::stop };
//this class implements the sensing unit of the system

class button_input {
protected:
    const static int MAX_SIZE = 5;  //max size is 5 input devices
    cmdEnum_push_button inputs[MAX_SIZE];   //array of digital input objects
    cmdEnum_push_button::cmdEnum player_input;  //stores the label of last recorded button press
    int buttons_used;
    bool init_flag;     //initialisation flag
public:
    button_input() { init_flag = false; }   //basic constructor
    button_input(int* pointer_to_pin_array, unsigned int longpush, int input_size) { ... }
    void setUpInputs(int* pointer_to_pin_array, unsigned int longpush, int input_size) { ... }
    bool isInitialised() { return init_flag; }
    bool isButtonPressed() {    //returns true if any button has been pressed
        if (init_flag) {    //if sensing unit initialised
            for (int i = 0; i < buttons_used; i++) {    //read all pins
                if (inputs[i].check_enabled()) {    //check if pin is initialised
                    if (inputs[i].check_n_get_new_input(player_input)) {    //check if button pressed, assign button label to player_input
                        return true;    //exit loop if button press detected
                    }
                }
            }
        }
        return false;//if no press detected, return false;
    }
    cmdEnum_push_button::cmdEnum getPlayerInput() { ... }
    int getButtonsUsed() { ... }
    void waitForInput() { ... }
};
```

Control unit

Task1_control.h class **Sensing**

Main functionality

- Keep track of score
- Determine match outcome
- Determine difficulty setting based on score
- Check if user input corresponds to generated sequence

Main functions

- checkInput() – takes two enumerator values as input and returns true if they match;
- isMatchOver() – returns true if match loss/win condition is met
- determinesDifficulty() – determines which difficulty level should be selected based on score

Additionally the control has functions to set, reset, update and print scores. The controller also sets the default match length.

```
class controller {
protected:
    int simon_score, player_score; //keeps track of score
    int match_length;//number of games to be played in match, excluding tie breaks
    const int default_match_length = 6;//default match length
public:
    controller() { ... }
    void setScores(int player, int simon) {
        simon_score = simon;
        player_score = player;
    }
    void resetScore() { ... }
    void updateScore(bool playerWin) { ... }
    bool checkInput(cmdEnum_push_button::cmdEnum expected, cmdEnum_push_button::cmdEnum player_input) {//checks if the player input is corre
        return (expected == player_input);//return true if input matches expected input
    }
    bool isMatchOver() {
        return ((simon_score + player_score == match_length) && (simon_score != player_score))
            ||((simon_score + player_score == match_length+1)&&(simon_score==match_length/2|| player_score == match_length / 2));
        //if match length reached and not a tie, or if match had a tiebreak;
    }
    bool didPlayerWin() {
        if (isMatchOver()) {//if match is over
            return player_score > simon_score;//and player scored more
        }
        else {
            return false;
        }
    }
    void printScore() { ... }
    int determineDifficulty() { ... }
};
```

EE3579: ACT 1; # *Krisjanis* # # *Visnevskis* #; # 5_1_7_6_8_1_9_7 #

Random unit

Task1_random.h class **simon_sequence**

Main functionality

- Generate random sequence of cues
- Store the random sequence to be used by other parts of the system

Main functions

- generateSequence() – generates a pseude random sequence of cues
- getSequence() – returns the array of generated sequences
- setButtonsUsed() – sets the number of buttons used for input, based on this the allowed cues for sequence generation change

```
1  #ifndef TASK1_RANDOM_H
2  #define TASK1_RANDOM_H
3  #include <Task1_sensing.h>
4  //this unit creates and stores the random sequence to be used in game
5  class simon_sequence {//
6      cmdEnum_push_button::cmdEnum sequence[100];//stores the sequence
7      int buttons_used;
8      bool init_flag;
9      void generateSequence() {//populates the sequence
10         unsigned long int seed = millis();
11         srand(seed);//set seed to be ms since board started running
12         for (int i = 0; i < 100; i++) {//populate enum array
13             sequence[i] = lables[rand() % buttons_used];//range [1, buttons_used]
14         }
15     }
16 public:
17     simon_sequence() {}//basic constructor
18     cmdEnum_push_button::cmdEnum* getSequence() { ... }
23     void setButtonsUsed(int numb_of_buttons) {
24         buttons_used = numb_of_buttons;
25         generateSequence();
26         init_flag = true;
27     }
28     ~simon_sequence() {}
29
```

System unit

Task1_system.h class **simon**

Main functionality

- Select game mode and difficulty
- Play the game
- Communicate with user through serial port
- Keep time. Detect if

Main functions

- playGame() – plays a game of required length
- playMatch() – plays a match consisting of games. Plays until match end condition is rached. Different playMatch() function for each gamemode (playMatchDefault(), playMatchHS, playMatchAdaptive())
- setDifficulty() – Sets the difficulty for the match

This unit is where the game comes together. All of the other units are interfaced with the system unit. It contains the main function that the arduiono sketch runs in the loop in order to play the game. The system unit also contains the timer object that detects if user input has come before response time has run out

```
10  class simon {
11  protected:
12      //variables
13      const int default_difficulty = 3, default_length = 5;//default values
14      int sequence_length;//sequence length
15      int cue_length;//cue length. length of buzz
16      bool result;//used to store if result has been reached
17      bool time_elapsed;//flag for if time has elapsed
18      cmdEnum_push_button::cmdEnum player_input;//stores player input
19      buzzer_unit output;//output object
20      button_input input;//input object
21      controller control;//control object
22      IntervalCheckTimer timer;//timer object for response time
23      int gamemode; //variable to hold gamemode
24      bool init_flag;
25      //functions
26      bool playGame(int length, cmdEnum_push_button::cmdEnum* sequence) { ... }
46      void playMatchDefault() { ... }
77      void playMatchHS() { ... }
96      void playMatchAdaptive() { ... }
129     void userSelectDifficulty() { ... }
150     void setDifficulty(int level) { ... }
162     void userSelectGamemode() { ... }
185 public:
186     simon() { ... }
190     simon(buzzer_unit buzzer, button_input button) { ... }
196     setUpSimon(buzzer_unit buzzer, button_input button) { ... }
204     void gameSelect() { ... }
214 };
```

General notes on software

The sequence of cues consists of a predefined set of enum values. These values are stored in the only global variable in this implementation called lables[]. In this implementation 5 of these values are defined, only allowing for a system with 5 input devices. However this can be easily expanded to an arbitrary number of input devices.

The random sequences generated consist of 100 cues, as it felt like a safe bet that this sequence length would never be reached by the player (I think an excellent player could maybe reach 50 on a good day). The modulo operator is used for populating the random sequence which is not ideal as it is known to introduce bias in the distribution. But the sequences generated are random enough and the details of pseudo random number generation are beside the point of this assignment anyway.

I have implemented also the option to pick between 3 game modes: default, high-score and adaptive. The selection of gamemode (and selection of difficulty for 2 of the gamemodes) is done through a serial “menu”. The menu implementation leaves much to be desired but it works just fine for this simple game. This section of the report is not very extensive as I am running out of time before submission but there are comments throughout the code that explain the general purpose and function of each function and variable.

2 Lower-Intermediate

Action module test

```
void testActionModule() {
	buzzer_unit test(5);//initialise buzzer object on pin 5
	//play all pre-set frequencies
	test.countdown();
	test.gameLoss();
	test.gameWin();
	test.matchLoss();
	test.matchWin();
	test.warning();
	//test the play_cue function
	for (int i = 0; i < 5; i++) {
		test.playCue(lables[i], false, 1000);//test cues
		test.playCue(lables[i], true, 1000);//test feedback
		delay(700);//need to add this delay because feedback tones have no delay after them as
defined
	}
	//test play_cue_sequence
	test.playCueSequence(lables, 5, 1000);
}
```

The action module test is pretty straight forward. The test function first initialises the buzzer unit on a pin (here pin 5 used as per HW layout). Then plays all the preset sequences. Then goes through and plays all five preset cues that correspond to lables[0] through lables[4]. The cues are played first as they would be played in a cue sequence and then as they would be played for feeback. Finally test the playCueSequence function passing the pointer to the array of lables corresponding to the preset tones as an argument. All tones played for 1000ms tone length (feedback tones will be half of that length).

Sensing module test

```
void testSensingUnit() {
	int pins[5] = { 2, 4, 7, 8 , 13};//pins to be used as inputs
	button_input test(pins, 150, 5);//initialise with pins in pins[], longpush 150ms, pins used 5
	if(test.isInitialised()){//if sensing unit initialised
		if (test.isButtonPressed()) {//if button press detected
			Serial.println("buttons used: " + (String)(test.getButtonsUsed()));//print
number of buttons used
			switch (test.getPlayerInput()) {//get which button was pressed
			case lables[0]:Serial.println("button 1 pressed"); break;//print which button
was pressed
			case lables[1]:Serial.println("button 2 pressed"); break;
			case lables[2]:Serial.println("button 3 pressed"); break;
			case lables[3]:Serial.println("button 4 pressed"); break;
			case lables[4]:Serial.println("button 5 pressed"); break;
			default: Serial.println("no input"); break;
			}//note that i have only 4 buttons, so to test if all 5 work i need to connect
one of the buttons to pin 13 here.
			test.waitForInput();//wait for user confirmation before recording next input
		}
	}
}
```

```
void testSensingUnit3Pins() {//same test but initialised using only 3 pins
	int pins[5] = { 2, 4, 7, 8, 13};//pins to be used as inputs
	button_input test(pins, 150, 3);//initialise with pins in pins[], longpush 150ms
	if (test.isInitialised()) {//if sensing unit initialised
		if (test.isButtonPressed()) {//if button press detected
			Serial.println("buttons used: " + (String)(test.getButtonsUsed()));//print
number of buttons used
			switch (test.getPlayerInput()) {//get which button was pressed
			case lables[0]:Serial.println("button 1 pressed"); break;//print which button
was pressed
			case lables[1]:Serial.println("button 2 pressed"); break;
			case lables[2]:Serial.println("button 3 pressed"); break;
			case lables[3]:Serial.println("button 4 pressed"); break;//shouldnt happen
here, no button 5 initialised
			case lables[4]:Serial.println("button 5 pressed"); break;//shouldnt happen
here, no button 5 initialised
			default: Serial.println("no input"); break;
			}//note that i have only 4 buttons, so to test if all 5 work i need to connect
one of the buttons to pin 13 here.
			test.waitForInput();//wait for user confirmation before recording next input
		}
	}
}
```

Two test functions implemented for the sensing unit, one with 3 input devices, one with 5. In both cases the sensing unit is initialised using an array of ints corresponging to pins to be used as input (pins 2, 4, 7, 8 and 13 here), and a longpush of 150ms.The first test function initialises 5 input devices, the 2n initialises 3 input devices. Then if a button press is detectes (ie. isButtonPressed()==true), print out the number of initialised input buttons, then which button has been pressed. Then wait for user input to continue. To test this just run the test and press the buttons a bunch to make sure everything is in order.

Sample output (testSensingUnit3Pins()):

buttons used: 3
button 1 pressed
buttons used: 3
button 2 pressed
buttons used: 3
button 3 pressed
buttons used: 3
button 3 pressed
buttons used: 3
button 2 pressed
buttons used: 3
button 2 pressed
buttons used: 3
button 1 pressed

And indeed the output corresponds to expected buttons. Button1 – pin 2, Button 2 – pin 4, Button 3 – pin 7. And when button on pin 8 is pressed, no output is detected.

Random module test

```
void testRandom() {//test with 4 buttons used
	simon_sequence test, test2;
	test.setButtonsUsed(4);
	cmdEnum_push_button::cmdEnum* sample_sequence = test.getSequence();
	for (int i = 0; i < 10; i++) {//print sequence
		switch (*(sample_sequence+i)) {
		case lables[0]:Serial.println("button 1"); break;
		case lables[1]:Serial.println("button 2"); break;
		case lables[2]:Serial.println("button 3"); break;
		case lables[3]:Serial.println("button 4"); break;
		case lables[4]:Serial.println("button 5"); break;
		}
	}
	Serial.println("");
	test2.setButtonsUsed(3);
	sample_sequence = test2.getSequence();
	for (int i = 0; i < 10; i++) {//print sequence
		switch (*(sample_sequence + i)) {
		case lables[0]:Serial.println("button 1"); break;
		case lables[1]:Serial.println("button 2"); break;
		case lables[2]:Serial.println("button 3"); break;
		case lables[3]:Serial.println("button 4"); break;
		case lables[4]:Serial.println("button 5"); break;
		}
	}
}
```

Declare 2 objects of class simon_sequence, initialise one to generate a sequence for 4 button game, the other for generating a sequence for a 3 button game. Then print which button the first 10 elements of each of those sequences corresponds to. Expect to see a random-ish sequence in the range [1,4] for first object and a random-ish sequence in the range [1,3] for the second object

```
Sample test output:

button 3
button 4
button 4
button 1
button 2
button 2
button 3
button 2
button 3
button 2

button 3
button 2
button 2
button 1
button 1
button 2
button 2
button 1
button 2
button 2
```

and indeed we do see expected test result

Control module test

```
void testController() {
	controller test;
	for (int i = 0; i < 7; i++) {//all possible player scores
		for (int j = 0; j < 7; j++) {//all possible simon scores
			test.setScores(i, j);//set the scores
			if (test.isMatchOver()) {//if match end condition met
				test.printScore();//print final score (expected 8 scores printed)
				Serial.println("Adaptive simon would choose difficulty:
"+(String)(test.determineDifficulty()));
				//print which difficulty level would be chosen at that score in an
adaptive difficulty match
				//expected higher difficulty when simon is losing, lower when simon is
winning. max when player at 3pts
			}//should print all possible score outcomes;
			if (test.didPlayerWin()) {
				Serial.println("player won!");
			}//4 player win cases expected
		}
	}
	int count = 0;//store number of matches found
	for (int i = 0; i < 5; i++) {//all possible sequence lables
		for (int j = 0; j < 5; j++) {//all possible sequence lables
			if (test.checkInput(lables[i], lables[j])) {//if the lables match
				count++;//increment match count
			}
		}
	}
	Serial.println((String)(count)+" matches found");//5 matches expected, (whenever i=j)
}
```

The control module is initialised when called. First test that isMatchOver() function is correct. Iterate through all possible player score and simon scores. Whenever a match win condition is met print out the final score. 8 scores printed expected. (6-0, 5-1, 4-2, 4-3, 3-4, 2-4, 1-5 and 0-6). Then test if didPlayerWin() functions properly. 4 of those decisive results should be player wins. Then test if the determineDifficulty() function works. We expect to see a lower difficult when simon is winning and higher difficulty when player is winning. With some exceptions: when simon has won for sure, the lowest difficulty will be selected (simon rubbing it in your face). And when a player is one game away from winning the match the highest difficult will be selected.

Sample test output:

```
player (0) - simon (6)
Adaptive simon would choose difficulty: 1
player (1) - simon (5)
Adaptive simon would choose difficulty: 1
player (2) - simon (4)
Adaptive simon would choose difficulty: 1
player (3) - simon (4)
Adaptive simon would choose difficulty: 5
player (4) - simon (2)
Adaptive simon would choose difficulty: 4
player won!
player (4) - simon (3)
Adaptive simon would choose difficulty: 4
player won!
player (5) - simon (1)
Adaptive simon would choose difficulty: 4
player won!
player (6) - simon (0)
Adaptive simon would choose difficulty: 4
player won!
5 matches found
Test output as expected
```

System module test

```
void test_system() {
	buzzer_unit action(5);//initialise action unit
	int pins[5] = { 2, 4, 7, 8, 13 };//pins for pushbuttons
	button_input sensing(pins, 200, 4);//initialise sensing unit
	simon test(action, sensing);//declare system unit with initialised action and sensing units
	test.gameSelect();// test if match plays properly, game results as expected, difficulty
selection works etc.
}
```

Now the system module test is not as straight forward as the other module tests. Here it is necessary for the user to test if game runs correctly and smoothly. First the action unit and input units are initialised. The random units will be initialised during the playMatch functions. The system unit test begins by calling the gameSlect() function. This is the main function that the system executes. It firsts prompts the user to select one of the three game modes using the input buttons. Then based on user selection the respective playMatch function is called. The playMatch functions can then either prompt the user to select difficult or select the difficult based on the match score. The playMatch functions also get the number of buttons used from the sensing unit and initialise the random unit using that value. The random sequence is then either generated at the start of each match or at the start of each game depending on game mode. In order to test thoroughly the system implementation the user goes through this checklist

- Ensure game mode selection works by selecting each of the game modes. (system doesn't get stuck at selection and appropriate game mode started after selection)
- Ensure that difficulty selection works by selecting each of the difficulties (system doesn't get stuck at selection and difficulty of the game changes appropriately). Only applies to default and high-score game modes. This is also a good step at which to tune the difficulty settings if desired.
- Ensure that adaptive difficulty behaves as expected
- Ensure that game win/loss occurs when the right condition is met. For each game mode check that game can be lost by either entering incorrect cue or by letting the response time elapse.

If no issues detected, the game doesn't get stuck or no unexpected wins or losses occur after a few test runs for each game mode, the test can be deemed a success;

2.1 Subsection (if needed)

3 Higher-Intermediate

A video “demo” showing the system working is shown: I do say in the video that sequence length is 4 cues, but I misspoke it is 5 cues.

To illustrate how the system functions I will walk through a sort of top-down program path of a default match being played. Below are the functions for playing a default match and playing a game;

```
void playMatchDefault() {//this function starts match
		userSelectDifficulty();//prompt user to select difficulty level for the match
		while (!control.isMatchOver()) {//while match end conditions not met
			simon_sequence new_sequence;//generate new sequence
			new_sequence.setButtonsUsed(input.getButtonsUsed());//initialises the sequence
			Serial.print("Current score: ");
			control.printScore();//print score to serial before game starts
			Serial.println("Are you rady for the next round? (press any button to start)");
			input.waitForInput();//halt game until user is ready
			output.countdown();//play the countdown
			output.playCueSequence(new_sequence.getSequence(), sequence_length,
cue_length);//play the cue sequence for this game
			result = playGame(sequence_length, new_sequence.getSequence());//play game,
game result stored in result flag
			control.updateScore(result);//update score
			if (result) {//if player wins game
				output.gameWin();//play game win song
			}
			else {//if player loses
				output.gameLoss();//play game loss song
			}
		}
		result = control.didPlayerWin();//if player wins assign true, simon wins assign false
		if (result) {//if player wins match
			output.matchWin();//play match win sequence
		}
		else {//if simon wins match
			output.matchLoss();//play match loss song
		}
		Serial.print("Final score: ");
		control.printScore();//prints final score
		control.resetScore();//reset scores to, can now start new game
	}
bool playGame(int length, cmdEnum_push_button::cmdEnum* sequence) {//plays a game of input length
		for (int i = 0; i < length; i++) {//check user input for each cue
			time_elapsed = false;//set time elapsed flag to false when checking each new
user input
			timer.updateCheckTime();//update check timer before each user input
			while (!time_elapsed) {//while time has not elapsed
				time_elapsed = timer.isMinChekTimeElapsed();//update time elapsed flag
				if (time_elapsed) { return false; }//if time elapsed, before input
detected exit game as a loss
				if (input.isButtonPressed()) {//check for user input button press
detected
					if (control.checkInput(sequence[i], input.getPlayerInput()))
{//check if input is correct
						output.playCue(sequence[i], true, cue_length);//play
feedback
						time_elapsed = true;//set time elapsed flag to true so
that next cue can be checked
					}
					else {//if incorrect input
						return false;//end game as a loss
					}
				}
			}
```

```
            }
            return true;
        }
```

In rough pseudocode to explain what the system does when a match start function is called:

playMatch()
Ask user to select difficulty:
Enter loop:
    Generate new sequence
    Print score
    Play cue sequence
    Play countdown
    Play game
    Update score;
    Play game win/loss sequence
    If Match finished, exit loop
Play match win/loss sequence
Reset score
Exit match

And the Play game function would be:

Play game():
Enter loop:
    Update timer
    Enter loop:
        Read input
        If input detected
            Play feedback
        If timer elapsed, exit loop
        If input detected, exit loop
    If game lost, exit loop as fail
    Else keep playing
    If all inputs correct exit loop
Record game result
Exit game

This is the backbone of how the system plays the game, broken down into the simplest components. Each module is responsible for executing a different part of these functions.
For example the sensing unit responsible for reading the input and reading player selection for the match difficult. The random unit is responsible for generating the new sequence.
The control unit determines when a game is lost or won, when a match is lost or won and printing scores. In one of my implemented game modes the control unit is also responsible for determining difficulty level.
The timer object provides an alternate game loss condition
The action unit is responsible for playing sequences that inform the player about the state of game, giving feedback on inputs, and playing the generated sequence.
The system unit is left with the responsibility of interfacing with all the other units so that they can work together to produce a game.
This is the simplest way I can think of to explain what the system does without getting into specifics about what each specific function does.

Each game mode has a slightly different match function but they are all based on the same game function

3.1 Subsection …

4 Advanced

4.1 Improving the System

INSTRUCTIONS (remove before submission): Once the previous sections have been completed, leading to the correct implementation, the Author agrees with the lecturer a set of improvements to the system (or to a subset of its components), leading to expended system functionality. The existing system (SW, HW) is modified accordingly. The author of the report explains the modifications to system units.
Include **code snippets**, **block diagrams, flow charts.** Include all contributed libraries and sketch files to the appendix.
Include **Screenshots and Pictures of the working system** to confirm it works as intended. Videos can be submitted along with the report: in this case, mention the relevant file name(s) in this section.

At the interview the Author will be asked question about the proposed implementation of the system and asked to sketch some modifications to the unit and to the system so it could perform a slightly different task.

5 Appendix (code)

Appendix for Section 1

//Task1_action.h file

```
#ifndef TASK1_ACTION_H
#define TASK1_ACTION_H
#include <Task1_sensing.h>
#include <Beginner_LED.h> //imcluded to visualise buzzer output
class buzzer_unit {
protected:
	bool init_flag;			//initialisation flag
	int buzzer_pin;			//pin for buzzer output
	const static int tones[5] = {392, 523, 660, 784, 100}; //tones for playing cues/feedback. not
changeable by user.
	dig_LED led[4];			//only used to visualize tones
public:
	buzzer_unit() {			//basic constructor
		init_flag = false;
	}
	buzzer_unit(int pin_number) {	//constructor with pin number
		buzzer_pin = pin_number;	//set buzzer pin
		init_flag = true;	//set initialised
		led[0].setup_LED(9); //just for visualisation
		led[1].setup_LED(10);
		led[2].setup_LED(11);
		led[3].setup_LED(12);
	}
	void setUpBuzzer(int pin_number) {//set up function
		if (!init_flag) {
			buzzer_pin = pin_number;	//set buzzer pin
			init_flag = true;	//set initialised
			led[0].setup_LED(9); //just for visualisation
			led[1].setup_LED(10);
			led[2].setup_LED(11);
			led[3].setup_LED(12);
		}
	}
	bool isInitialised() { return init_flag; }
	//next section is all the different tone sequences, they are set to specific lengths, not
intended to be changable by user
	void countdown() { //3 med tones followed by high tone signify start of game, large delay
between
		if (init_flag) {//chek if initialised
			tone(buzzer_pin, 559, 500); delay(800);
			tone(buzzer_pin, 559, 500); delay(800);
			tone(buzzer_pin, 559, 500); delay(800);
			tone(buzzer_pin, 745, 500); delay(800);
		}
		else { Serial.println("Buzzer not initialised"); }
	}
	void gameWin() { //sequence to be played when game is won rising melody
		if (init_flag) {//check if initialised
			tone(buzzer_pin, 392, 1000); delay(500);
			tone(buzzer_pin, 440, 1000); delay(500);
			tone(buzzer_pin, 523, 1000); delay(500);
			tone(buzzer_pin, 659, 1000); delay(1500);
		}
		else{ Serial.println("Buzzer not initialised"); }

	}
	void gameLoss() { //falling melody
		if (init_flag) {
			noTone(buzzer_pin);//added to stop feedback from playing
```

```
                        tone(buzzer_pin, 659, 1000); delay(500);
                        tone(buzzer_pin, 523, 1000); delay(500);
                        tone(buzzer_pin, 440, 1000); delay(500);
                        tone(buzzer_pin, 392, 1000); delay(1500);
                }
                else{ Serial.println("Buzzer not initialised"); }
        }
        void matchWin() { //"We are the champions" by queen
                if (init_flag) {
                        noTone(buzzer_pin);
                        tone(buzzer_pin, 587, 1500); delay(1500);
                        tone(buzzer_pin, 554, 1000); delay(1000);
                        tone(buzzer_pin, 587, 500); delay(500);
                        tone(buzzer_pin, 554, 1500); delay(1500);
                        tone(buzzer_pin, 494, 2000); delay(2000);
                }
                else { Serial.println("Buzzer not initialised"); }
        }
        void matchLoss() { //ominous/sad melody
                if (init_flag) {
                        tone(buzzer_pin, 349, 1000); delay(1000);
                        tone(buzzer_pin, 349, 1000); delay(1000);
                        tone(buzzer_pin, 349, 1000); delay(1000);
                        tone(buzzer_pin, 277, 2000); delay(2000);
                }
                else{ Serial.println("Buzzer not initialised"); }
        }
        void warning() { //Three high tones
                if (init_flag) {
                        tone(buzzer_pin, 698, 1000); delay(1000);
                        tone(buzzer_pin, 698, 1000); delay(1000);
                        tone(buzzer_pin, 698, 1000); delay(1000);
                }else{ Serial.println("Buzzer not initialised"); }
        }
        void playCue(cmdEnum_push_button::cmdEnum button, bool isFeedback, int cue_length) { //same
function used to play cue and feedback

                if (init_flag) {
                        int tone_length = cue_length, time_delay = cue_length * 1.2;
                        if (isFeedback) {//if a feedback is being played, reduce the tone length
                                tone_length = cue_length / 2;
                                time_delay = 0;//when feedback is being provided, dont wait for one
feedback to end before starting next
                        }
                        switch (button) {
                        case lables[0]:
                                noTone(buzzer_pin); delay(50);//make sure there is gap between feedbacks
                                tone(buzzer_pin, tones[0], tone_length);
                                led[0].switch_on();
                                delay(time_delay);
                                led[0].switch_off();
                                break;
                        case lables[1]:
                                noTone(buzzer_pin); delay(50);//make sure there is gap between feedbacks
                                tone(buzzer_pin, tones[1], tone_length);
                                led[1].switch_on();
                                delay(time_delay);
                                led[1].switch_off();
                                break;
                        case lables[2]:
                                noTone(buzzer_pin);
                                delay(50);//make sure there is gap between feedbacks
                                tone(buzzer_pin, tones[2], tone_length);
                                led[2].switch_on();
                                delay(time_delay);
                                led[2].switch_off();
                                break;
                        case lables[3]:
```

```
                        noTone(buzzer_pin); delay(50);//make sure there is gap between feedbacks
                        tone(buzzer_pin, tones[3], tone_length);
                        led[3].switch_on();
                        delay(time_delay);
                        led[3].switch_off();
                        break;
                case lables[4]:
                        noTone(buzzer_pin); delay(50);//make sure there is gap between feedbacks
                        tone(buzzer_pin, tones[4], tone_length);
                        delay(time_delay);
                        break;
                }
        }
        else { Serial.println("Buzzer not initialised"); }
        //the one big problem with the play_cue() function is that when you get a sequence of
same tones, and you
        //press the buttons quickly enough, the feedbacks overlap, so you dont get a sense of
how many times you
        //have pressed the same button

    }
    void playCueSequence(cmdEnum_push_button::cmdEnum* sequence, int sequence_length, int
cue_length) {//plays an array containing sequence
        if (init_flag) {
                for (int i = 0; i < sequence_length; i++) {//go through array that holds
sequence and play all the cues
                        playCue(sequence[i], false, cue_length);
                }
        }
    }

};

void testActionModule() {
    buzzer_unit test(5);//initialise buzzer object on pin 5
    //play all pre-set frequencies
    test.countdown();
    test.gameLoss();
    test.gameWin();
    test.matchLoss();
    test.matchWin();
    test.warning();
    //test the play_cue function
    for (int i = 0; i < 5; i++) {
        test.playCue(lables[i], false, 1000);//test cues
        test.playCue(lables[i], true, 1000);//test feedback
        delay(700);//need to add this delay because feedback tones have no delay after them as
defined
    }
    //test play_cue_sequence
    test.playCueSequence(lables, 5, 1000);
}

#endif
```

Appendix for Section 2

//Task1_sensing.h file

```
#ifndef TASK1_SENSING_H
#define TASK1_SENSING_H

#include "CmdEnumPushButton.h"

constexpr cmdEnum_push_button::cmdEnum lables[5] = { cmdEnum_push_button::button1,
cmdEnum_push_button::button2,
cmdEnum_push_button::button3, cmdEnum_push_button::button4, cmdEnum_push_button::stop };
//this class implements the sensing unit of the system

class button_input {
protected:
	const static int MAX_SIZE = 5;	//max size is 5 input devices
	cmdEnum_push_button inputs[MAX_SIZE];	//array of digital input objects
	cmdEnum_push_button::cmdEnum player_input;	//stores the label of last recorded button
press
	int buttons_used;
	bool init_flag;		//initialisation flag
public:
	button_input() { init_flag = false; }	//basic constructor
	button_input(int* pointer_to_pin_array, unsigned int longpush, int input_size) {
	//constructor with arguments
		for (int i = 0; i < input_size; i++) {	//go through array of pushbutton objects
			inputs[i].assign_pin_command(*(pointer_to_pin_array + i), lables[i],
longpush);//assigns pin numbers and labels to buttons
		}
		buttons_used = input_size;//number of buttons used
		init_flag = true;	//set initialisation flag to true
	}
	void setUpInputs(int* pointer_to_pin_array, unsigned int longpush, int input_size) {//set up
inputs with pin array and longpush length
		if (!init_flag) {
			for (int i = 0; i < input_size; i++) {
				inputs[i].assign_pin_command(*(pointer_to_pin_array + i), lables[i],
longpush);//assigns pin numbers and labels to buttons
			}
			buttons_used = input_size;//number of buttons used
			init_flag = true;
		}
	}
	bool isInitialised() { return init_flag; }
	bool isButtonPressed() {	//returns true if any button has been pressed
		if (init_flag) {	//if sensing unit initialised
			for (int i = 0; i < buttons_used; i++) {  //read all pins
				if (inputs[i].check_enabled()) {  //check if pin is initialised
					if (inputs[i].check_n_get_new_input(player_input)) {	//check if
button pressed, assign button label to player_input
						return true;  //exit loop if button press detected
					}
				}
			}
		}
		return false;//if no press detected, return false;
	}
	cmdEnum_push_button::cmdEnum getPlayerInput() {//returns last user input
		if (init_flag) {
			return player_input;
		}

	}
	int getButtonsUsed() {
		if (init_flag) {
			return buttons_used;
		}
	}
```

```
	void waitForInput() {//function to wait for user input before proceeding
		if (init_flag) {
			while (!isButtonPressed()) {//stop waiting when any button pressed
			//waiting for user confirmation to proceed
			}
		}
	}
};

//Testing functions

void testSensingUnit() {
	int pins[5] = { 2, 4, 7, 8 , 13};//pins to be used as inputs
	button_input test(pins, 150, 5);//initialise with pins in pins[], longpush 150ms, pins used 5
	if(test.isInitialised()){//if sensing unit initialised
		if (test.isButtonPressed()) {//if button press detected
			Serial.println("buttons used: " + (String)(test.getButtonsUsed()));//print
number of buttons used
			switch (test.getPlayerInput()) {//get which button was pressed
			case lables[0]:Serial.println("button 1 pressed"); break;//print which button
was pressed
			case lables[1]:Serial.println("button 2 pressed"); break;
			case lables[2]:Serial.println("button 3 pressed"); break;
			case lables[3]:Serial.println("button 4 pressed"); break;
			case lables[4]:Serial.println("button 5 pressed"); break;
			default: Serial.println("no input"); break;
			}//note that i have only 4 buttons, so to test if all 5 work i need to connect
one of the buttons to pin 13 here.
			test.waitForInput();//wait for user confirmation before recording next input
		}
	}
}
void testSensingUnit3Pins() {//same test but initialised using only 3 pins
	int pins[5] = { 2, 4, 7, 8, 13};//pins to be used as inputs
	button_input test(pins, 150, 3);//initialise with pins in pins[], longpush 150ms
	if (test.isInitialised()) {//if sensing unit initialised
		if (test.isButtonPressed()) {//if button press detected
			Serial.println("buttons used: " + (String)(test.getButtonsUsed()));//print
number of buttons used
			switch (test.getPlayerInput()) {//get which button was pressed
			case lables[0]:Serial.println("button 1 pressed"); break;//print which button
was pressed
			case lables[1]:Serial.println("button 2 pressed"); break;
			case lables[2]:Serial.println("button 3 pressed"); break;
			case lables[3]:Serial.println("button 4 pressed"); break;//shouldnt happen
here, no button 5 initialised
			case lables[4]:Serial.println("button 5 pressed"); break;//shouldnt happen
here, no button 5 initialised
			default: Serial.println("no input"); break;
			}//note that i have only 4 buttons, so to test if all 5 work i need to connect
one of the buttons to pin 13 here.
			test.waitForInput();//wait for user confirmation before recording next input
		}
	}
}
#endif
```

//Task1_control.h file

```
#ifndef TASK1_CONTROL_H
#define TASK1_CONTROL_H
#include <Task1_sensing.h>//included for cmdEnum definiton
//this class keeps track of score, checks if input correct, checks if match is over
class controller {
protected:
	int simon_score, player_score; //keeps track of score
	int match_length;//number of games to be played in match, excluding tie breaks
	const int default_match_length = 6;//default match length
public:
	controller() {//if no value specified initialise to default values
		simon_score = 0;
		player_score = 0;
		match_length = default_match_length;
	}
	void setScores(int player, int simon) {
		simon_score = simon;
		player_score = player;
	}
	void resetScore() {
		simon_score = 0;
		player_score = 0;
	}
	void updateScore(bool playerWin) {
		if (playerWin) { //if playerWin = true, increment plater score
			player_score++;
		}
		else { //otherwise increment simon score;
			simon_score++;
		}
	}
	bool checkInput(cmdEnum_push_button::cmdEnum expected, cmdEnum_push_button::cmdEnum
player_input) {//checks if the player input is correct
		return (expected == player_input);//return true if input matches expected input
	}
	bool isMatchOver() {
		return ((simon_score + player_score == match_length) && (simon_score != player_score))
			||((simon_score + player_score ==
match_length+1)&&(simon_score==match_length/2|| player_score == match_length / 2));
		//if match length reached and not a tie, or if match had a tiebreak;
	}
	bool didPlayerWin() {
		if (isMatchOver()) {//if match is over
			return player_score > simon_score;//and player scored more
		}
		else {
			return false;
		}
	}
	void printScore() {//prints score to serial
		Serial.println("player (" + (String)(player_score) + ") - simon (" +
(String)(simon_score)+")");
	}
	int determineDifficulty() {//for adaptive difficulty setting
		if (player_score == match_length/2) { return 5; }//if player about to win, set max
difficulty
		if (simon_score == match_length / 2 + 1) { return 1; }//if simon has won, set easiest
difficulty. this is simon milking his victory
		switch (player_score - simon_score) {
		case 0: return 3; break;//if currently a tie, set medium difficulty
		case 1: return 4; break;//if simon losing by 1 set to 2nd hardest
		case -1: return 2; break;//if simon winning by one set to 2nd easiest
		case -2: return 1; break;//if simon winning by 2 set to easiest
		}
		if (player_score - simon_score > 1) { return 4; }//if player winning by more than 1,
set to 2nd hardest
```

```
            if (player_score - simon_score < -2) { return 1; }//if player losing by more than 2,
set to easiest difficulty
            //this function probably should be rewritten in a proper way
      }
};

//Test functions

void testController() {
      controller test;
      for (int i = 0; i < 7; i++) {//all possible player scores
            for (int j = 0; j < 7; j++) {//all possible simon scores
                  test.setScores(i, j);//set the scores
                  if (test.isMatchOver()) {//if match end condition met
                        test.printScore();//print final score (expected 8 scores printed)
                        Serial.println("Adaptive simon would choose difficulty:
"+(String)(test.determineDifficulty()));
                        //print which difficulty level would be chosen at that score in an
adaptive difficulty match
                        //expected higher difficulty when simon is losing, lower when simon is
winning. max when player at 3pts
                  }//should print all possible score outcomes;
                  if (test.didPlayerWin()) {
                        Serial.println("player won!");
                  }//4 player win cases expected
            }
      }
      int count = 0;//store number of matches found
      for (int i = 0; i < 5; i++) {//all possible sequence lables
            for (int j = 0; j < 5; j++) {//all possible sequence lables
                  if (test.checkInput(lables[i], lables[j])) {//if the lables match
                        count++;//increment match count
                  }
            }
      }
      Serial.println((String)(count)+" matches found");//5 matches expected, (whenever i=j)
}
#endif
```

//Task1_random.h file

```
#ifndef TASK1_RANDOM_H
#define TASK1_RANDOM_H
#include <Task1_sensing.h>
//this unit creates and stores the random sequence to be used in game
class simon_sequence {//
	cmdEnum_push_button::cmdEnum sequence[100];//stores the sequence
	int buttons_used;
	bool init_flag;
	void generateSequence() {//populates the sequence
		unsigned long int seed = millis();
		srand(seed);//set seed to be ms since board started running
		for (int i = 0; i < 100; i++) {//populate enum array
			sequence[i] = lables[rand() % buttons_used];//range [1, buttons_used]
		}
	}
public:
	simon_sequence() {}//basic constructor
	cmdEnum_push_button::cmdEnum* getSequence() {//return pointer to the sequence
		if (init_flag) {
			return(sequence);
		}
	}
	void setButtonsUsed(int numb_of_buttons) {
		buttons_used = numb_of_buttons;
		generateSequence();
		init_flag = true;
	}
	~simon_sequence() {}

};

//test

void testRandom() {//test with 4 buttons used
	simon_sequence test, test2;
	test.setButtonsUsed(4);
	cmdEnum_push_button::cmdEnum* sample_sequence = test.getSequence();
	for (int i = 0; i < 10; i++) {//print sequence
		switch (*(sample_sequence+i)) {
		case lables[0]:Serial.println("button 1"); break;
		case lables[1]:Serial.println("button 2"); break;
		case lables[2]:Serial.println("button 3"); break;
		case lables[3]:Serial.println("button 4"); break;
		case lables[4]:Serial.println("button 5"); break;
		}
	}
	Serial.println("");
	test2.setButtonsUsed(3);
	sample_sequence = test2.getSequence();
	for (int i = 0; i < 10; i++) {//print sequence
		switch (*(sample_sequence + i)) {
		case lables[0]:Serial.println("button 1"); break;
		case lables[1]:Serial.println("button 2"); break;
		case lables[2]:Serial.println("button 3"); break;
		case lables[3]:Serial.println("button 4"); break;
		case lables[4]:Serial.println("button 5"); break;
		}
	}
}
#endif
```

//Task1_system.h

```
#ifndef TASK1_SYSTEM_H
#define TASK1_SYSTEM_H

#include <Task1_sensing.h>
#include <Task1_action.h>
#include <Task1_control.h>
#include <Task1_random.h>
#include <IntervalCheckTimer.h>

class simon {
protected:
	//variables
	const int default_difficulty = 3, default_length = 5;//default values
	int sequence_length;//sequence length
	int cue_length;//cue length. length of buzz
	bool result;//used to store if result has been reached
	bool time_elapsed;//flag for if time has elapsed
	cmdEnum_push_button::cmdEnum player_input;//stores player input
	buzzer_unit output;//output object
	button_input input;//input object
	controller control;//control object
	IntervalCheckTimer timer;//timer object for response time
	int gamemode; //variable to hold gamemode
	bool init_flag;
	//functions
	bool playGame(int length, cmdEnum_push_button::cmdEnum* sequence) {//plays a game of input
length
		for (int i = 0; i < length; i++) {//check user input for each cue
			time_elapsed = false;//set time elapsed flag to false when checking each new
user input
			timer.updateCheckTime();//update check timer before each user input
			while (!time_elapsed) {//while time has not elapsed
				time_elapsed = timer.isMinChekTimeElapsed();//update time elapsed flag
				if (time_elapsed) { return false; }//if time elapsed, before input
detected exit game as a loss
				if (input.isButtonPressed()) {//check for user input button press
detected
					if (control.checkInput(sequence[i], input.getPlayerInput()))
{//check if input is correct
						output.playCue(sequence[i], true, cue_length);//play
feedback
						time_elapsed = true;//set time elapsed flag to true so
that next cue can be checked
					}
					else {//if incorrect input
						return false;//end game as a loss
					}
				}
			}
		}
		return true;
	}
	void playMatchDefault() {//this function starts match
		userSelectDifficulty();//prompt user to select difficulty level for the match
		while (!control.isMatchOver()) {//while match end conditions not met
			simon_sequence new_sequence;//generate new sequence
			new_sequence.setButtonsUsed(input.getButtonsUsed());//initialises the sequence
			Serial.print("Current score: ");
			control.printScore();//print score to serial before game starts
			Serial.println("Are you rady for the next round? (press any button to start)");
			input.waitForInput();//halt game until user is ready
			output.countdown();//play the countdown
			output.playCueSequence(new_sequence.getSequence(), sequence_length,
cue_length);//play the cue sequence for this game
			result = playGame(sequence_length, new_sequence.getSequence());//play game,
game result stored in result flag
			control.updateScore(result);//update score
```

```
                        if (result) {//if player wins game
                                output.gameWin();//play game win song
                        }
                        else {//if player loses
                                output.gameLoss();//play game loss song
                        }
                }
                result = control.didPlayerWin();//if player wins assign true, simon wins assign false
                if (result) {//if player wins match
                        output.matchWin();//play match win sequence
                }
                else {//if simon wins match
                        output.matchLoss();//play match loss song
                }
                Serial.print("Final score: ");
                control.printScore();//prints final score
                control.resetScore();//reset scores to, can now start new game
        }
        void playMatchHS() {//one sequence generated at start of match, pllayer tries to get the
highest score possible
                simon_sequence new_sequence;//generate new sequence
                new_sequence.setButtonsUsed(input.getButtonsUsed());//initialises the sequence
                sequence_length = 1;//start with sequence length 1
                userSelectDifficulty();
                result = true;//kind of like 0th game was won
                while (result) {
                        Serial.println("Current score: " + (String)(sequence_length - 1));//print
current high score
                        Serial.println("Are you rady for the next round? (press any button to start)");
                        input.waitForInput();//halt game until user is ready
                        output.countdown();
                        output.playCueSequence(new_sequence.getSequence(), sequence_length,
cue_length);//play the cue sequence for this round
                        result = playGame(sequence_length, new_sequence.getSequence());
                        if (result) { output.gameWin(); }
                        sequence_length++;
                }
                output.gameLoss();
                Serial.println("final score: " + (String)(sequence_length - 2));//print current high
score
        }

        void playMatchAdaptive() {//in this gamemode the difficulty changes based on score
                while (!control.isMatchOver()) {//while match end conditions not met
                        Serial.println("we got to here");
                        setDifficulty(control.determineDifficulty());//control determines what
difficulty level to select at start of game
                        simon_sequence new_sequence;//generate new sequence
                        new_sequence.setButtonsUsed(input.getButtonsUsed());//initialises the sequence
                        Serial.print("Current score: ");
                        control.printScore();//print score to serial before game starts
                        Serial.println("Are you rady for the next round? (press any button to start)");
                        input.waitForInput();//halt game until user is ready
                        output.countdown();//play the countdown
                        output.playCueSequence(new_sequence.getSequence(), sequence_length,
cue_length);//play the cue sequence for this game
                        result = playGame(sequence_length, new_sequence.getSequence());//play game,
game result stored in result flag
                        control.updateScore(result);//update score
                        if (result) {//if player wins game
                                output.gameWin();//play game win song
                        }
                        else {//if player loses
                                output.gameLoss();//play game loss song
                        }
                }
                result = control.didPlayerWin();//if player wins assign true, simon wins assign false
                if (result) {//if player wins match
```

```
                        output.matchWin();//play match win sequence
                }
                else {//if simon wins match
                        output.matchLoss();//play match loss song
                }
                Serial.print("Final score: ");
                control.printScore();//prints final score
                control.resetScore();//reset scores to, can now start new game

        }
void userSelectDifficulty() {//prompt user to select difficulty
                int difficulty = default_difficulty;//set difficulty to default
                bool confirmed = false;//flag for if the difficulty selection is confirmed
                Serial.println("Please select difficulty: (button1:++  button2:-- button3:
confirm)");//give user instructions for difficulty selection
                Serial.println("Default difficulty: " + (String)(difficulty));//print default
difficulty
                while (!confirmed) {//while user hasnt confirmed difficulty selection
                        input.waitForInput();//wait for an input
                        switch (input.getPlayerInput()) {//when input received decide how to change
difficulty
                        case lables[0]://if user pressed button with lable "1"
                                if (difficulty < 5) { difficulty++; } break;// increase diff, make sure
the difficulty doesnt go out of bounds
                        case lables[1]:
                                if (difficulty > 1) { difficulty--; } break;// decrease diff, make sure
the difficulty doesnt go out of bounds
                        case lables[2]:
                                confirmed = true; break; //if button3 pressed, confirm difficulty
setting
                                //if button4 or button 5 pressed do nothing
                        }
                        Serial.println("Slected difficulty: " + (String)(difficulty));//print
difficulty selection
                }
                setDifficulty(difficulty);
                Serial.println("Difficulty set, have fun!");//notify user that difficulty selected and
match will begin
        }
        void setDifficulty(int level) {//preset difficulty levels in range [1,5]
                int t_resp;//resposne time for each input
                switch (level) {//based on selected difficulty level
                case 1: t_resp = 2000; break;//very easy
                case 2: t_resp = 1500; break;
                case 3: t_resp = 1000; break;
                case 4: t_resp = 800; break;
                case 5: t_resp = 500; break;//very difficultf
                }
                timer.setInterCheck(t_resp);//set timer to check in an interval of t_reso
                cue_length = t_resp / 2;//cue lengths made shorter than t_resp for nicer playing
experience
        }
        void userSelectGamemode() {
                gamemode = 1;//set gamemode to default
                bool confirmed = false;
                Serial.println("Please select gamemode: (button1:++  button2:-- button3:
confirm)");//give user instructions for gamemode selection
                Serial.println("Available gamemodes:");//print gamemode options
                Serial.println("1 - Default");//game mode as required per guidelines
                Serial.println("2 - High score");//like the original simon says
                Serial.println("3 - Default but simon's difficulty is adaptive");// simon will do his
best to win the game (adaptive difficulty)
                while (!confirmed) {//while user hasnt confirmed diffficulty selection
                        input.waitForInput();//wait for an input
                        switch (input.getPlayerInput()) {//when input received decide how to change
difficulty
                        case lables[0]://if user pressed button with lable "1"
                                if (gamemode < 3) { gamemode++; } break;// increase gamemode
```

```
                        case lables[1]:
                                if (gamemode > 1) { gamemode--; } break;// decrease gamemode
                        case lables[2]:
                                confirmed = true; break; //if button3 pressed, confirm difficulty
setting
                                //if button4 or button 5 pressed do nothing
                        }
                        Serial.println("Slected gamemode: " + (String)(gamemode));//print difficulty
selection
                }
                Serial.println("Gamemode set, have fun!");//notify user that difficulty selected and
match will begin
        }
public:
        simon() {
                init_flag = false;
                sequence_length = default_length;
        }
        simon(buzzer_unit buzzer, button_input button) {
                output = buzzer;//assign output module to be used
                input = button;//assign input module to be used
                sequence_length = default_length;//set sequence length to default length
                init_flag = output.isInitialised() && input.isInitialised();//controller always
initialised when called so no need to check
        }
        setUpSimon(buzzer_unit buzzer, button_input button) {
                if (!init_flag) {
                        output = buzzer;//assign output module to be used
                        input = button;//assign input module to be used
                        sequence_length = default_length;//set sequence length to default length
                        init_flag = output.isInitialised() && input.isInitialised();//controller always
initialised when called so no need to check
                }
        }
        void gameSelect(){//main function that system executes
                if (init_flag) {
                        userSelectGamemode();
                        switch (gamemode) {
                        case 1: playMatchDefault(); break;
                        case 2: playMatchHS(); break;
                        case 3: playMatchAdaptive(); break;
                        }
                }
        }
};

//Test function.

void test_system() {
        buzzer_unit action(5);//initialise action unit
        int pins[5] = { 2, 4, 7, 8, 13 };//pins for pushbuttons
        button_input sensing(pins, 200, 4);//initialise sensing unit
        simon test(action, sensing);//declare system unit with initialised action and sensing units
        test.gameSelect();// test if match plays properly, game results as expected, difficulty
selection works etc.
}
#endif
```

Appendix for Section 3

Appendix for Section 4