

**School of Engineering**

Academic Year: 2021/22

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:** | [# *Piotr* #] |
| **Family Name:** | [# *Rucinski* #] |
| **Student ID:** | [# 5\_1\_9\_8\_5\_9\_6\_1 #] |
| **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:** 08 March 2022 |

**SCHOOL OF ENGINEERING**

**PLAGIARISM AWARENESS DECLARATION**

**Course Code ……… EE3579 …………**

**SURNAME/FAMILY NAME: ………………**# *Rucinski* #**…………………………………..**

**FIRST NAME: ……………**# *Piotr* #**……………………..**

**ID Number: ……………**# 5\_1\_9\_8\_5\_9\_6\_1 #**………………….**

**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*](http://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:\_\_\_\_\_\_\_**# *Piotr* # **\_**# *Rucinski* #**\_\_\_\_**

**Date:\_\_\_\_\_\_\_\_\_\_**08 March 2022**\_\_\_\_\_\_\_\_\_\_\_\_**

**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. Marks up to 6 CGS (if confirmed at the interview)**

**LOW-INTERMEDIATE: Section 2. Marks up to 6 CGS (if confirmed at the interview)**

**HIGH-INTERMEDIATE: Section 3.** **Marks up to 5 CGS (if confirmed at the interview)**

**ADVANCED: Section 4. Marks up to 5 CGS (if confirmed at the interview)**

Contents

[Student details 1](#_Toc97597257)

[Plagiarism Awareness Declaration Form 2](#_Toc97597258)

[1 Basic (max 5 pages) 4](#_Toc97597259)

[1.1 Overview of the system and its units – input & output 4](#_Toc97597260)

[1.2 Input unit 4](#_Toc97597261)

[1.3 Output unit 6](#_Toc97597262)

[1.4 Testing of the units 8](#_Toc97597263)

[2 Lower-Intermediate (max 5 pages) 9](#_Toc97597264)

[2.1 Overview of the control unit 9](#_Toc97597265)

[2.2 Testing of the control unit 10](#_Toc97597266)

[2.3 Game class 11](#_Toc97597267)

[3 Higher-Intermediate (max 5 pages) 13](#_Toc97597268)

[3.1 Expanding on the system with new features 13](#_Toc97597269)

[3.2 Playing the Advanced Game 14](#_Toc97597270)

[4 Advanced 15](#_Toc97597271)

[4.1 Overview of the task – proximity sensor 15](#_Toc97597272)

[4.1 Coding the system 16](#_Toc97597273)

[4.2 Playing the game with the proximity sensor 18](#_Toc97597274)

[5 Appendix (code) 19](#_Toc97597275)

# Basic (max 5 pages)

## Overview of the system and its units – input & output

System selections, based on student ID:

|  |  |  |
| --- | --- | --- |
| **System Behaviour** | **Relevant student ID (5th, 6th, 7th or 8th)** | **Selection** |
| Cue Device | 8th - 1 | Buzzer – up to 5 frequencies supported |
| Game and match score | 6th – 9 (odd) | Match spans 6 games + tiebreaker |
| Game clock | 5th – 5 | Clock expires if **all** cues take longer |
| Game outcome | 8th – 1 (in 0-4) | Game ends after L player responses |
| Difficulty selection | 7th – 6 | Potentiometer |

Overall, the Simon Says system has been split into three units: Control Unit, Input Unit and an Output Unit. This section will focus on the Input and Output units. Both these units are closely connected to the hardware and the Fritzing circuit layout shown below will be used as the most elementary building block of the entire system.

Diagram

Description automatically generated

**Figure 1**. Elementary system built for testing input & output.

## Input unit

Now I will describe the input unit. It uses the Basic\_input.h, provided via the labs. It has a number of functions responsible for the correct working of the unit. The class diagram is shown below.

Graphical user interface, application

Description automatically generated

**Figure 2**. The schematics of the input unit.

The variables that control the flow of the unit are presented below. What is interesting in this design is that it’s core is array based – and that precisely fits the design of the control unit (described later), which allows for little to no “adapting” to the other unit, as most operations are index-based. Based off of how the user spins the potentiometer, there are 5 distinct difficulty levels that can be chosen. The feedback parameter is passed into the control unit so that user gets real-time feedback of the difficulty level.

Text

Description automatically generated

**Figure 3**. Contents of the inputUnit class.

Text

Description automatically generatedText

Description automatically generatedI would like to focus on the most interesting function of this class – readBinaryInputs(). While perhaps slightly overengineered, it will work for any sort of input device (be it a button or a Hall Effect sensor). It will be triggered only once – on a falling edge of a press. The contents of the binaryInputs array are read through the returnPressedButton function, which is responsible for communication with the Output unit, so that it knows which sound to play. Both functions are shown below.

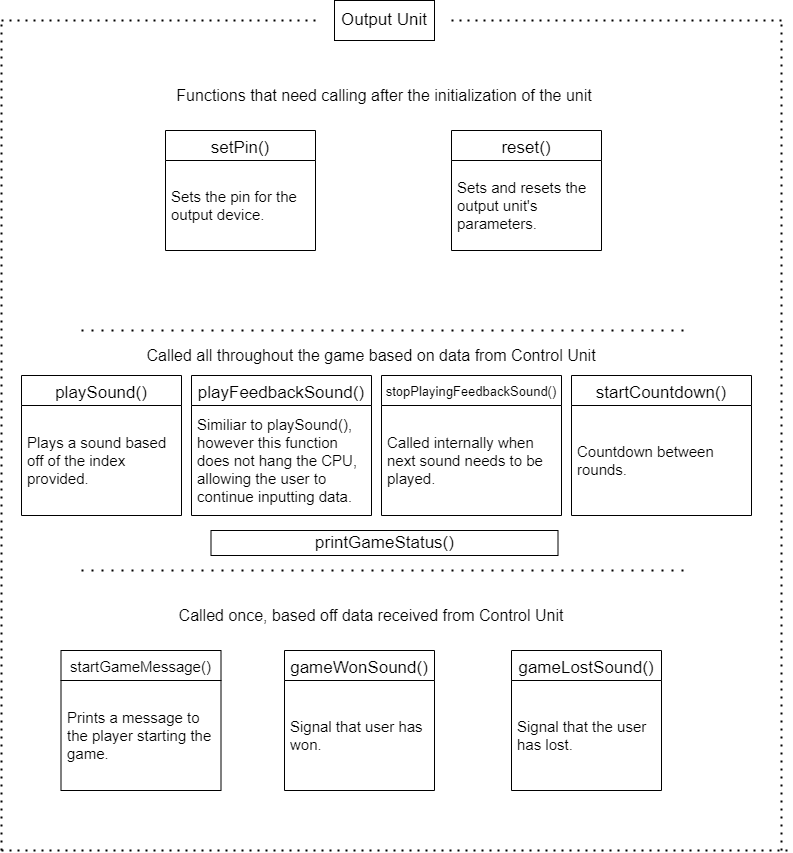
**Figure 4**. readBinaryInputs() and returnPressedButton() functions.

For further inspection, please refer to the files attached to the report: **SSInputUnit.h**.

## Output unit

Output unit is based on a simple **Buzzer.h** class that is attached to the document. This buzzer class is mostly a utility one – it combines plenty of functions into a few easier ones so that the code is more understandable on the Output level. The Piezo Buzzer can generate sounds of different frequencies, and those are supported by the system.

The output unit itself is responsible for generating sounds, as well as pushing messages to Serial where necessary, just for that increased readability. It is worth noting that a person that has played the game at least once will know how the game works can enjoy it without the terminal. I believe it is worth mentioning that this is not the best implementation that could have been created, as it has one fatal flaw – this Output Unit will only ever work with one game being played at a time – so, in case the user wanted to play two games in parallel (on an Arduino Mega, for example), they would be unable to do so. The reason why is that for a lot of the sounds that are being played via the output unit, the programme simply hangs the CPU for a certain amount of time. This is perhaps a bad practice, but it is sufficient for this case and definitely helpful, as this unit does not have to communicate with the control unit regarding timing at all. The class diagram is shown below.



**Figure 5**. Class diagram of the Output Unit.

For the frequencies, I have decided to make them musical notes, as that is sure to increase the pleasantness of the game. When the Output Unit receives an index from the Input Unit, it simply chooses that frequency to play. The details of communications between fragments will be discussed in the next section. For inspection, please refer to the file **SSOutputUnit.h**.

Text

Description automatically generated

**Figure 6**. Internal variables of the class outputUnit. Ignore bool advanced for now as it will be explored in a later section.

## Testing of the units

Testing has been done through an Arduino sketch – please refer to **SimonSaysBasicTest.ino.** Most of the functions (mainly those that do not require any control) have been tested there and passed. A screenshot of the tests conducted can be viewed below. Please also see video **PRBasicTests.mp4** for the testing routine in action.

Text

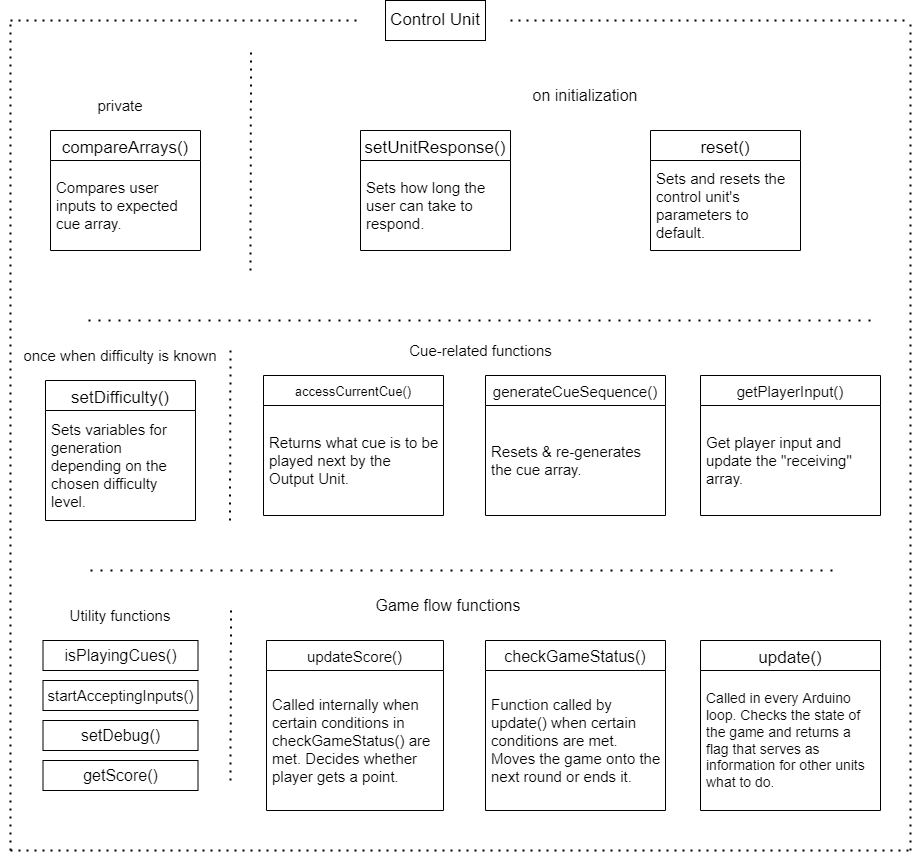
Description automatically generated

**Figure 7**. Test results on the Serial Monitor.

# Lower-Intermediate (max 5 pages)

## Overview of the control unit

The control unit is the heart of the game. This is the component that has no hardware representation and is run purely on the Arduino, sending out signals to be picked up by all the other components.



**Figure 8**. Class diagram of the Control Unit.

The control unit, through the use of flags in update(), keeps track of what should be happening in this Arduino loop. It returns the following statuses: -1: player lost, 0 – move on to the next round, 1 – player won, 2 – stay in the same round. There is room for improvement with this encoding as encoding this as an int is fairly memory inefficient, however even making it a short int still requires the same amount of memory. Either way, these flags are very important in telling the other units what to do – and it is primarily through this function that it communicates with the rest of the components. For further inspection, please see **SSControlUnit.h.** Below is the internal data of a Control Unit object, as well as a closer look into the logic behind the update() function.

Text

Description automatically generated

**Figure 9**. Internal data of controlUnit class.

Text

Description automatically generated

**Figure 10**. update() function. The weirdFlag is certainly eye catching, but it was necessary for “return checkGameStatus()” not to be triggered on the first loop, effectively losing the player a round.

## Testing of the control unit

The control unit was tested via **SimonSaysControl.ino** file. Please see the attached **PRControlTests.mp4** video.

The case of testing these units is interesting in the sense that having it work properly served as the basis for the creation of the Game class that puts all the units together, as it would be hard to test it otherwise with the way it was designed. Results of one run are shown below.

Text, letter

Description automatically generatedTable

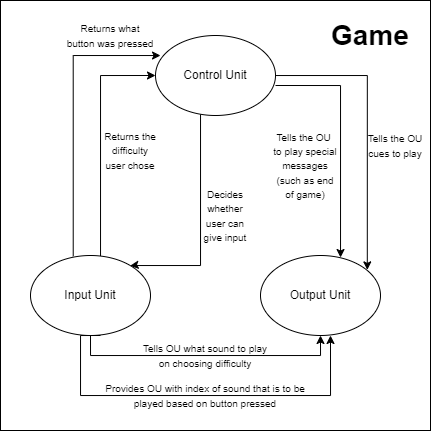
Description automatically generatedText, table

Description automatically generated

**Figure 11**. Testing of the Control Unit. See how different scenarios are explored (e. g. not pressing all buttons, pressing wrong buttons, tiebreaker game)

## Game class

Having completed testing, it was natural to create a Game class out of the tests that have been conducted to clean up the code a bit, as well as help with inheritance in the next section. Game class is very simple in nature – it contains three functions – setupGame() that is responsible for setting up all the components, setDifficulty() that takes care of setting the difficulty and gameLoop(), which is meant to be run every iteration of the Arduino loop. The explanation of the code is within the class itself, so please do inspect **Game.h**. A diagram exploring the communication between the 3 units is shown below. The sketch running the game is **SimonSaysGame.ino**.



**Figure 12**. Communications between units, handled inside the Game class.

Text

Description automatically generated

**Figure 13**. Class structure of the game class.

# Higher-Intermediate (max 5 pages)

## Expanding on the system with new features

With the base game completed, now many of the features have been expanded. To demonstrate my ability when it comes to using inheritance, I have created a new class – **SSAdvancedControl.h** which inherits from SSControlUnit. A lot of the features were easily implementable – and therefore the class can be easily fit into 30 lines of code. The only method that needs overloading is the setDifficulty() method. It has been overloaded with an additional argument – playTimeModifier that is passed by reference. Now, depending on the difficulty chosen, the player will have less time to respond in game, as well as the cues will be played faster. What is more, upon choosing difficulty, the time it took the user to choose will set a seed for the RNG – meaning that a user will virtually always play a different match of different cues.

Text

Description automatically generated

**Figure 14.** Using inheritance to create an Advanced Control Unit.

In my design, I have decided not to use inheritance to expand the Output Unit. The reason for that is very simple – I had to add only two lines of code to adapt that class for the advanced version. I have also unintentionally implemented some features prematurely (such as response on a higher frequency) for my basic version – therefore I found it redundant to rebuild the class just to force inheritance. The two lines simply set the mode to advanced and change the frequency to a higher one, and the other function, setRespTime, will be called in the AdvancedGame upon selecting difficulty.

No features require any changes to the Input Unit either and so it remains the same.

Text

Description automatically generated

Figure 15. Snippet of outputUnit class. These functions set the advanced features.

After finishing the advanced classes, they were put into **AdvancedGame.h.** AdvancedGame inherits from Game and, as advised by Dr Verdicchio, attempted to do pointer manipulation so as to minimize copy + paste of code. Unfortunately, my attempts were fruitless and I decided my time was better spent on Section 4. A good practice would have been to have AdvancedGame overload only setDifficulty() method – however, that would require swapping over a controlUnit object with an advancedControlUnit one. Since I was unsuccessful, I unfortunately had to go through the 2 minute struggle of replacing the control unit with the advanced one – something I was very unsatisfied with. Below is the AdvancedGame class, with slightly modified setDifficulty() function.

Text

Description automatically generated

**Figure 16**. advancedGame inheriting from game. Note how variable “modifier” is passed by reference.

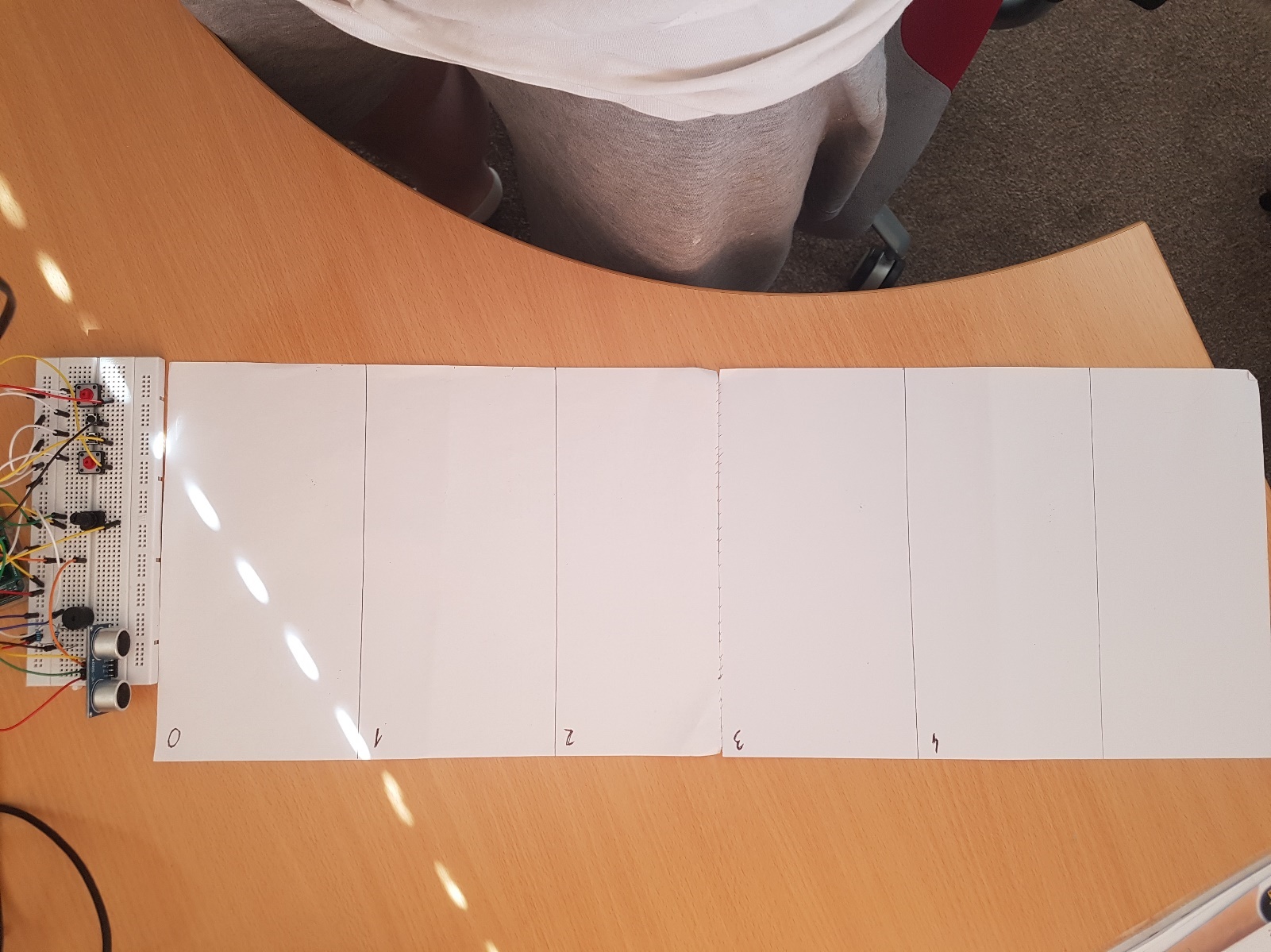
## Playing the Advanced Game

No formalized testing has been done for a very simple reason – and that is this will run exactly the same as the tests for the base game. Thanks to it compact structure, one can simply run **SimonSaysAdvancedGame.ino.** A video has been attached to the report demonstrating a playthroughon the highest difficulty – please see **PRAdvancedGame.mp4**.

# Advanced

## Overview of the task – proximity sensor

For the report, I have been asked to implement a version of the game that involves the ultrasound sensor. The ultrasound sensor should detect user input based on the distance to the sensor and then, on the press of a button, the user input would be captured. I have created a helper game mat for the player that shows where an object should be [perpendicularly] placed. These zones are approximately 10 cm apart – however, in code these values have been defined based off precise measurements from the ultrasound sensor. The “game mat” and overall system schematics is shown below.



**Figure 17**. Picture of the game mat. Notice how I decided not to dismantle the other buttons – they are still there, but they will be disconnected. The mat is not necessary to play the game, but serves as a reference point to the user.

Diagram

Description automatically generated

**Figure 18**. Fritzing layout of the circuit. Additional buttons are unnecessary, and the distance sensor unit has been added. In reality, my Arduino is on the other side of the breadboard – this is just for demonstration’s purpose.

## Coding the system

Having tested the circuit **with the provided Ultrasound4pinsTest.ino** file, I have moved onto implementing a class that could handle the new input unit. There is 1 new command – takeMeasurement(), and plenty of commands have been overloaded to save on memory. All in all, very few new variables have been introduced and all existing components have been used to lesser or bigger extent (such as using only the first element of the arrays). As this is a fairly interesting class, I have pasted snippets of all functions in. Please see **ProximityInput.h** for reference.

Text

Description automatically generated

**Figure 19**. Constructor and takeMeasurement().

Text

Description automatically generated

**Figure 20**. setDigitalPins() overload.

Text

Description automatically generated

**Figure 21**. readBinaryInputs() overload.

Text

Description automatically generated

**Figure 22**. returnPressedButton() overload. The name is not very fitting, however overloading this ensures that no other structure needs to be changed outside this class.

I have also needed to make a minor adjustment to the OutputUnit class to ensure that the user gets baseline extra time (4 times as much) to respond to cues, as this involves physically moving around an object. Since the modifier will be greater than 1, I have added the line that this needn’t apply to the speed at which the cues are being played.



**Figure 23**. Change made to setRespTime(). The change is uninvasive thanks to the “if” statement.

In the end, I have implemented another version of the game in a file called **ProximityGame.h.** Yet again, pointer manipulation could have probably been used to simplify the class greatly. Unfortunately I was unable to do so and therefore there was plenty of copy and paste. Biggest notable changes to AdvancedGame class are: overloading setupGame() to take an array of pin numbers of size 3 instead of 5, overloading setDifficulty() to increase the response time fourfold (note how on the highest difficulty, it will be 2.4 base time). With pointer manipulation, gameLoop() would not have to be overloaded, as the design handles everything inside of **ProximityInput.h,** so there would be no unexpected behaviors.

## Playing the game with the proximity sensor

Yet again, no testing routine is required. The game can be played in sizes up to 5, however it works best at up to 4. The game can be played through **SimonSaysProximityGame.ino**. Please see attached videos **ProximityGame1.mp4** and **ProximityGame2.mp4**, showcasing a single “round” on the highest difficulty. Text

Description automatically generated with medium confidence

**Figure 24**. Sample output on the console. Instead of “Button pressed”, there should be “input detected” – however, I have not changed that inside the OU class. The game is very hard at the highest difficulty level.

# Appendix (code)

Appendix for Section 1

SimonSaysBasicTest.ino

Buzzer.h

SSInputUnit.h

SSOutputUnit.h

Basic\_Input.h

PRBasicTests.mp4

Appendix for Section 2

SimonSaysControl.ino

SimonSaysGame.ino

SSControlUnit.h

Game.h

PRControlTests.mp4

Appendix for Section 3

SimonSaysAdvancedGame.ino

SSAdvancedControl.h

AdvancedGame.h

PRAdvancedGame.mp4

Appendix for Section 4

SimonSaysProximityGame.ino

Ultrasound4pinsTest.ino

UltrasoundSensor.h

ProximityInput.h

ProximityGame.h

PRProximityGame1.mp4

PRProximityGame2.mp4