Software Design
Project Puzzlebox

Thomas in 't Anker, Loek Le Blansch, Lars Faase, Elwin Hammer

Version 0.0, 2024-04-01: draft

Table of Contents

List of Figures

List of Tables

1. Introduction

This document contains all the design considerations made for the separate software components that make up the puzzle box. This document has a top-down structure, and has three levels of design 'depth':

1. Top-level (hardware diagrams, OSes, communication buses, etc.)
2. Module-level (puzzle inputs/outputs, top-level software diagram, etc.)
3. Component-level (software dependencies, game state, etc.)

Only design details deemed relevant by the document authors are documented here. Low-level implementation details such as API interfaces, code paths and workarounds are documented with Doxygen [1].

2. Top-Level

This section of the design document establishes the development target hardware. It also specifies the modules that are elaborated further in Section 3.

Figure 1 shows a block diagram that describes the context in which the puzzle box is used. The puzzle box in this diagram internally consists of a main controller and multiple puzzle modules. Other notable details include:

- The charger is removable, and the puzzle box is intended to be used as a battery-powered device (i.e. not while tethered).
- Puzzle outputs are used to complete a feedback loop (gameplay) with the players, as well as eventually provide a solution to diffuse a bomb. This bomb is part of a standalone project that has already been finished at the time of writing (2024-03-11), and this project only describes the interface between the puzzle box and the bomb.
- The puzzle box is capable of bidirectional communication over Wi-Fi. This connection is used to configure the puzzle box before gameplay or modify its state during gameplay (R-167, R-168).

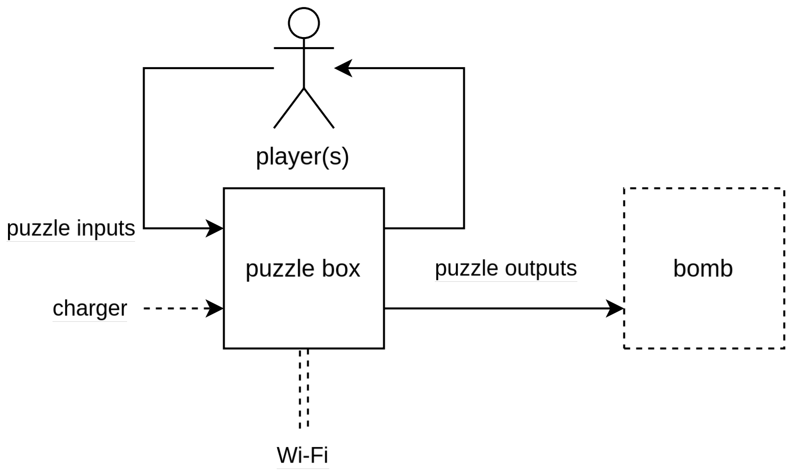

Figure 1. Context block diagram

The rest of this section details the internal hardware modules and the separation of functionality of these modules.

2.1. Puzzle Modules

The puzzle box hardware produced by the 21-22 group consists of 6 sides, 4 of which are utilized by puzzle modules. This section defines the properties of a puzzle module.

Puzzle modules can occupy one or more physical sides of the puzzle box to implement the physical interface required for a puzzle or game. In order to realize a complete game, each of these puzzle modules must have the ability to control game inputs and outputs. Two approaches for this were considered:

1. Let the main controller handle game state and logic for all puzzle modules.

   This approach has the main benefit of allowing puzzle module controllers to be substituted for I^{2}C I/O expanders that draw less power than a complete MCU.

   The major drawback of this approach is that the main controller's software needs to be configured for each different hardware configuration, which makes the main controller software more complicated.

2. Design an abstract 'game interface' and give each puzzle module its own MCU.

   This approach provides the most flexibility, as the main controller's software is no longer dependent on the physically installed hardware. This approach is also favorable with regards to testability, as each puzzle module can run standalone.

   The main drawback of this approach is the possible increase in power consumption, as each puzzle module now must have its own MCU for managing game state and communication with the main controller.

The current hardware (developed by the 21-22 group) uses the second approach, though with MCUs that were not designed for power efficiency. This year (23-24), the hardware produced by the 21-22 group was utilized due to the 23-24 group not including students from the hardware study path. Minimizing power draw for each puzzle module is still a priority, so a different microcontroller was selected.

The criteria for a puzzle module controller are:

- Must have an I^{2}C peripheral (R-141).
- Should have enough I/O ports to directly control moderately complex puzzles (R-142).
- Should be power efficient (R-143).

The research document [2] compares various microcontrollers matching these criteria. As a result of this research, the Microchip PIC16F15276 was selected as the recommended microcontroller for future puzzle modules.

Note | The current development hardware still utilizes an ESP32-PICO-D4 module, but due to a misunderstanding [3], Arduino boards were used to implement the puzzle modules.

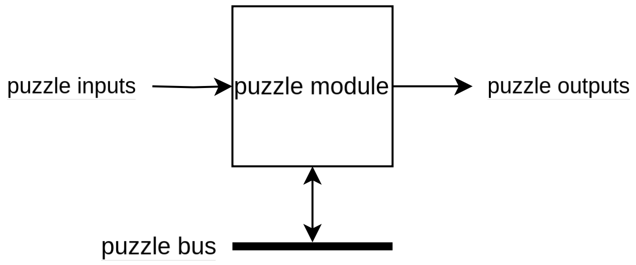

Figure 2. Generic puzzle module top-level block diagram

Figure 2 shows a block diagram of how most puzzle modules are implemented. Since the internal components of the puzzle module block from Figure 2 differ for each puzzle, they are left out in this section. Section 3 includes the next detail level for all of the implemented puzzles this year. The puzzle bus is detailed further in Section 2.3.

2.2. Main Controller

This section describes the responsibilities of the main controller inside the puzzle box. The main controller is a central processor that is responsible for the following:

- Integrate installed puzzle modules to form a cohesive experience by—
  - Detecting and initializing installed puzzle modules.
  - Aggregating game state for all installed puzzle modules.
  - Reading and writing game state for all installed puzzle modules.
  - Broadcasting updates internally.
- Serve a TCP socket for—
  - Sending state updates
  - Manually updating game state
  - Integration with the bomb

The specific requirement of being able to serve TCP socket connections was created so this year's puzzle box could keep compatibility with the software produced by the 21-22 group.

As mentioned in the research document [2], the 21-22 group produced the hardware that is used as development target for this year's (23-24) run of the puzzle box project. The existing hardware utilizes a Raspberry Pi 3B+ as main controller, but this controller caused issues with power consumption [4]. Choosing a different controller during development requires significant refactoring, so a different main controller has been selected at the start of this year's run of the puzzle box project.

The criteria for the main controller are:

- Must have an I^{2}C peripheral (R-136).
- Must be able to connect to a standard 802.11b/g/n access point (R-137).
- Must be able to serve TCP socket connection(s) (R-138).
- Should be power efficient (R-166).

The requirements document compares various microcontrollers that fit these criteria. After this comparison, the decision was made to utilize the Raspberry Pi Pico W as main controller during development.

Note | This was written while we did not know the puzzle bus specifically requires slave-addressible I^{2}C multi-master controllers to function properly. The RP2040 was still used, but has required implementing workarounds. Please see the handover report for more details on how this impacted the project [3].

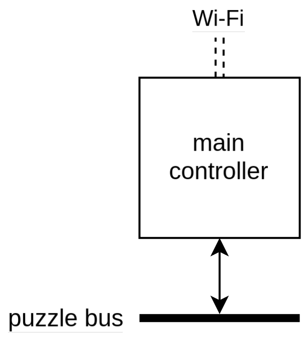

Figure 3. Main controller top-level block diagram

Figure 3 shows a block diagram of the main controller and its inputs and outputs. The main controller is the only module in the puzzle box that is able to communicate over Wi-Fi and is therefore responsible for all communication between the puzzle box and game operator. The puzzle bus is detailed further in Section 2.3.

2.3. Communication

Communication between puzzle modules, the main controller and other auxiliary peripherals is handled through a central I^{2}C bus referred to as the 'puzzle bus'. This design was again carried over from the hardware design from the 21-22 group [5].

The previously specified "HarwareInterrupt" line1[1] has been removed this year, as it was connected but not utilized [2].

To optimize for flexibility and extensibility, the puzzle bus should ideally function as a network (similar to the CAN bus), to allow puzzle modules to send responses asynchronously. I^{2}C was initially chosen for the puzzle bus due to its widespread availability on microcontrollers and in peripherals, but does not provide any network-like capabilities.

To archive network-like capabilities, the puzzle bus is specified to be a multi-master I^{2}C bus, and all puzzle modules are specified to be I^{2}C multi-master controllers that are slave-addressible. The multi-master part is required to prevent I^{2}C transmissions from being corrupted in the event of a bus collision, and the slave-addressible part is required to both send and receive messages on the same controller. This can also be achieved by using 2 I^{2}C peripherals on the same bus simultaniously, which is what the RP2040 currently uses. This has required changes to the wiring, and is the only hardware-level specification made this year.

More details on the messages sent over the puzzle bus are described in Section 3.2.2.

2.4. Power supply

One of the user requirements is that the puzzle box runs on battery power (R-089). Due to the team composition of this year's (23-24) run of the puzzle box project, a new power supply was not chosen, even though the current power supply was determined insufficient by the 21-22 group. This year, additional requirements were specified for the power supply, which were used when selecting MCUs suitable for battery-powered applications.

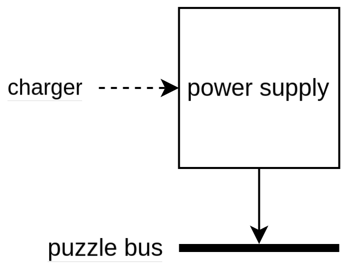

Figure 4. Power supply module top-level block diagram

Figure 2 shows a block diagram of how most puzzle modules are implemented. Besides the additional requirements, the power supply remains the same, and will not be elaborated further on in this document.

2.5. Overview

Figure 5 is the resulting combination of the modules from Figure 2, Figure 3 and Figure 4.

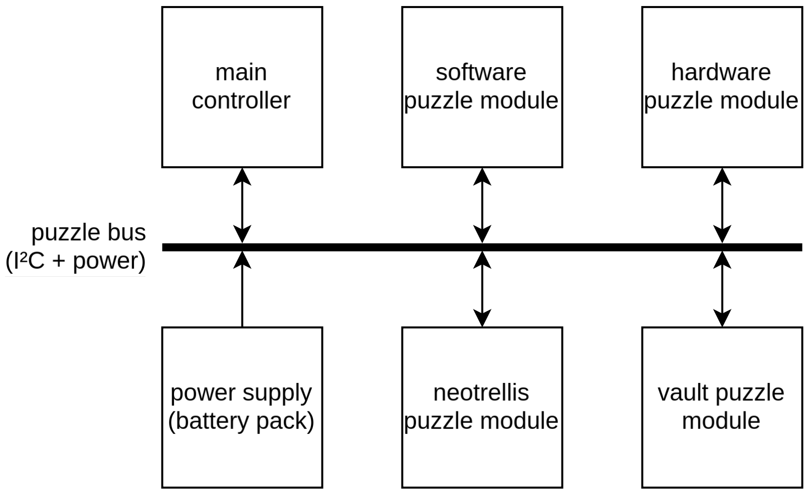

Figure 5. Hardware component overview

3. Modules

This section elaborates on the top-level (hardware) specifications from Section 2 with software design decisions.

3.1. Software module separation

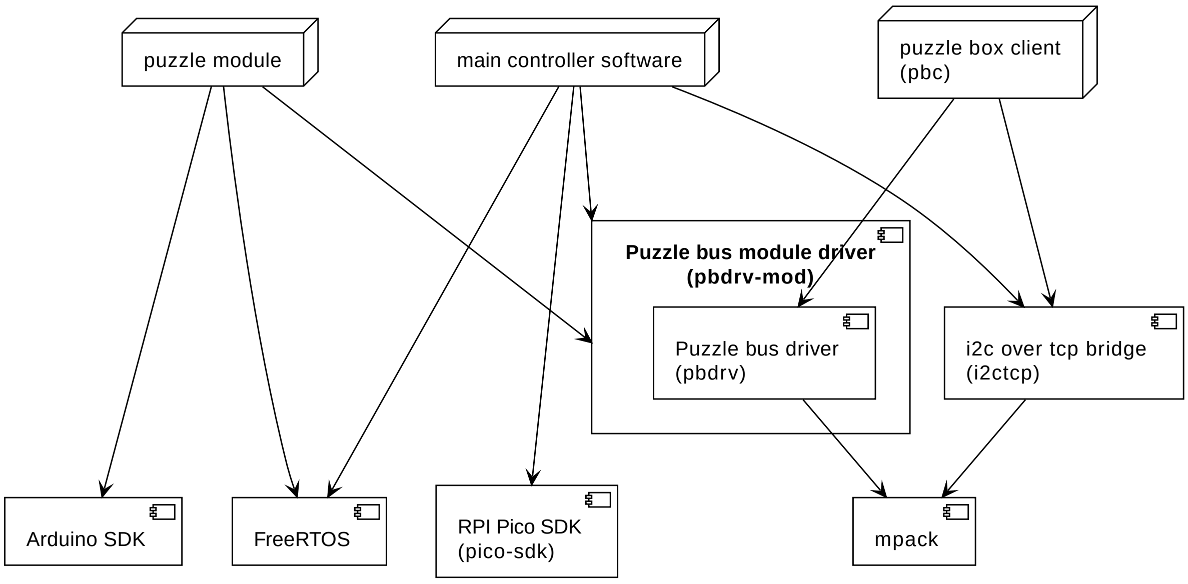

Figure 6. Software library components

Figure 6 shows a software component diagram with an example Arduino-based puzzle module, the main controller and the puzzle box client. Notable properties of this architecture include:

- The Arduino SDK, FreeRTOS, mpack, and RPI Pico SDK are external libraries, and with the exception of mpack, these have not been modified.
- The puzzle bus driver is split into a (portable) standalone library and a module-specific driver.
- There is a separate library for (de)serializing I^{2}C commands for transmission over TCP.

The specific decision to split the puzzle bus driver and create a separate I^{2}C over TCP library was made to avoid writing a command set separate from internal puzzle bus commands (i.e. one specific to TCP connections). This architecture allows the puzzle box client to not only control the main controller, but also directly inspect puzzle bus messages for debugging and development of future puzzle modules (R-130).

3.2. Puzzle module framework

This subsection defines aspects of the 'puzzle framework': the interface that allows puzzle modules to integrate with the puzzle bus and main controller. All communication described within this subsection refers to 'internal' communication between the main controller and puzzle modules on the puzzle bus.

The puzzle framework is the foundation of the puzzle box software, and is designed to facilitate the following:

- Allow puzzle modules to be swapped with minimal downtime or maintenance (R-132).
- Simplify the development process and integration of new puzzle modules (R-130).
- Provide abstracted interfaces to allow for easy integration of the puzzle box as part of a larger whole (R-133).

3.2.1. Guidelines

The following assumptions are made about puzzle modules:

- Puzzle modules do not take initiative to send REQ or SET commands. They only respond to requests from the main controller.
- Puzzle modules are I^{2}C multi-master controllers that are slave-addressable in master mode.
- The main controller is a puzzle module, but with the following differences:
  - The main controller is allowed to initiate REQ and SET commands.
  - The main controller’s global state is an aggregation of all other puzzle modules, and ignores STATE SET commands.

These guidelines allow the following simplifications:

- Puzzle modules may assume they are always being addressed by the main controller (this simplifies the message architecture).
- The puzzle bus may be shared with regular I^{2}C peripherals without causing issues.

3.2.2. Messages

Puzzle bus messages consist of a simple header and variable data. This format is shown in Table 1. The messages are (de)serialized using mpack. This choice was made after considering various alternative options for sending structured messages [2]. Note that all messages are sent as I^{2}C writes. Due to this, the I^{2}C address of a message sender is included in the header to facilitate network-like features over I^{2}C.

Field	Content
type	Command type (see Table 2)
action	Command action (see Table 3)
sender	I^{2}C address of sender
cmd	Command data (dependent on type)

Table 1. Puzzle bus message format

Table 2 lists the different command types.

Type	Description
MAGIC	The MAGIC command effectively serves as a 'secret handshake' (using a *magic* value) which is used to distinguish between puzzle modules and unrelated I^{2}C devices.
STATE	The STATE command is used by puzzle modules to inform the main controller about their global state (see Section 3.2.3). The main controller aggregates the states of all connected puzzle modules and exchanges this aggregated state with the puzzle modules to indicate when the entire puzzle box is solved.
PROP	The PROP command type is used for exchanging arbitrary data between puzzle modules and/or the puzzle box client (pbc) over the TCP bridge. These properties are not used by the puzzle framework, and serve as an extensible interface for puzzle module developers to use.

Table 2. Puzzle bus command types

Table 3 lists the different command actions.

REQ	Mark the command as a request. The receiver of this message is expected to return a message of the same type, but with a RES action.
RES	Mark the command as a response to a REQ message. All REQ messages should be followed by a RES message.
SET	Request a change / write. The SET command is never followed by a RES.

Table 3. Puzzle bus command actions

Please note that not all commands define behavior for all actions (e.g. there is no MAGIC SET command).

Only the MAGIC and STATE commands are required for the puzzle box to function. The PROP command was created to allow future expansion without modifying the main controller firmware (R-130).

The specific format of the 'cmd' field from Table 1 is different for each command type, and is for this reason only documented in-code using Doxygen [1].

3.2.3. State

All puzzle modules implement the same state machine shown in Figure 7. Note that continuous arrows indicate state transitions that a puzzle module may take on its own, while dashed arrows indicate state transitions forced by the main controller. The main controller also allows the game operator to manually set a module’s global state to one of these states, which can be used to skip a puzzle if a player is stuck (R-168) or reset a game if it is malfunctioning (R-167).

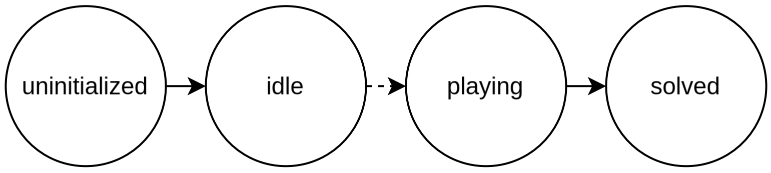

Figure 7. Global puzzle module state machine

Puzzle modules start in the 'uninitialized' state, where they wait until the main controller sends a REQ STATE command. Receiving this command indicates to the puzzle module that it was successfully registered by the main controller, and that it may transition from the 'uninitialized' state to the 'idle' state. This process is also shown in Figure 8.

The state machine described in Figure 7 is referred to as the global state. Puzzle modules may also declare and define custom variables, which are referred to as properties. These properties may contain game-specific variables; e.g. the binary state of each button on the Neotrellis puzzle module, or the last passcode entered on the vault puzzle module.

Separating properties from the global state allows the main controller to handle these property values as an arbitrary blob, which allows for future expansion without modification of the main controller software (R-130).

3.2.4. I^{2}C addresses

The RPI Pico SDK prohibits the use of I^{2}C addresses reserved by the I^{2}C specification. This means different addresses from previous years are used. These addresses are indexed in the code in a header exposed by the puzzle bus driver [1].

The I^{2}C addresses are also used to determine the puzzle sequence (i.e. the order in which puzzle modules are set to the 'playing' state). The sequence is determined by the main controller on startup, and consists of the connected puzzle modules' addresses in descending order (i.e. highest address first). Note that the same I^{2}C address may be used by two different puzzle modules, but this will make it impossible for them to be used simultaniously.

3.3. Main Controller

This subsection defines the function and state of the main controller.

3.3.1. Initializing puzzle modules

The main controller sends a MAGIC REQ command to every I^{2}C address on startup. Puzzle modules start in the 'uninitialized' state (see Figure 7), during which they do nothing. Puzzle modules in this state are still able to reply to requests, including MAGIC REQ commands. When the main controller receives a MAGIC RES command, the I^{2}C address of the sender is added to an internal list of I^{2}C devices that are considered puzzle modules.

After the initial handshake request 'wave' (bus scan), all puzzle modules are repeatedly asked for their global state using a STATE REQ command. This request also includes the global state of the requesting puzzle module, which is always the main controller (under normal circumstances). Upon receiving the first STATE REQ command, a puzzle module knows it has been registered successfully by the main controller, and transitions into the 'idle' state.

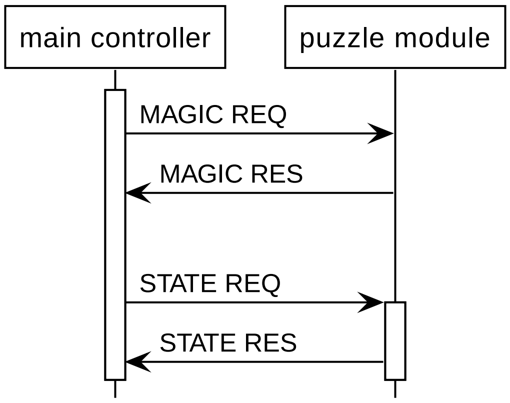

Figure 8. Puzzle module initialization sequence diagram

(Activated lifeline indicates the module is no longer in 'uninitialized' state)

3.3.2. State

The global state of the main controller is an aggregation of all installed puzzle modules and is defined by the state machine shown in Figure 9.

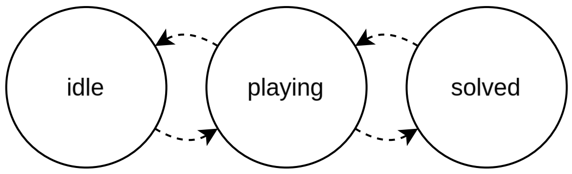

Figure 9. Main controller global state machine

The main controller global state is determined using the following rules:

- If all puzzle modules are in the 'idle' state, the main controller is also in the 'idle' state.
- If all puzzle modules are in the 'solved' state, the main controller is also in the 'solved' state.
- Else, the main controller is in the 'playing' state.

Due to the repeated STATE REQ commands, this effectively informs the puzzle modules when the puzzle box is completely solved.

3.3.3. Bridge

The bridge is used to remotely access and control the puzzle box.

The Raspberry Pi 3B+ used as main controller during the 21-22 run of the project set up a Wi-Fi Mesh network [5] to communicate with the puzzle box. This year's main controller (Raspberry Pi Pico W [2]) uses a standard 802.11b/g/n access point instead (R-137) as it is a simpler solution.

On this network, the main controller hosts a server that serves a TCP socket connection. This socket is used to directly forward all internal messages sent on the puzzle bus to the puzzle box client bidirectionally (on behalf of the main controller).

Due to the separation of the puzzle bus driver code into a standalone library for reading/writing puzzle bus commands, and a puzzle module-specific code, the puzzle box client is able to read/write raw I^{2}C commands directly. A separate library was made for serializing I^{2}C messages so they can be sent over the TCP connection. This library is documented in detail using Doxygen [1].

3.3.4. Operating system

Because the main controller needs to asynchronously handle state exchanges with puzzle modules while serving a TCP socket connection, the decision to use a task scheduler was made. Due to the requirement that most software should be covered by the standard curriculum (R-169), this choice was between FreeRTOS and Zephyr. FreeRTOS was chosen because it is the simplest solution, and because the features Zephyr offers over FreeRTOS are already present in the Raspberry Pi Pico SDK.

3.4. NeoTrellis puzzle

This subsection defines aspects of the 'NeoTrellis puzzle' module and gives a summary of how the puzzle is meant to be solved. This module will be created to facilitate the NeoTrellis puzzle game logic and communication with the main controller about the puzzle state.

3.4.1. NeoTrellis puzzle gameplay

The NeoTrellis puzzle is an 8x8 button matrix with Neopixels underneath each button. The way to solve this puzzle is by dimming every Neopixel in the 8x8 matrix. This is done by clicking on a button, which switches the state of the Neopixel underneath the pixel and the Neopixels in each cardinal direction from the pressed button. This means that if a Neopixel was on and the button was pressed it will turn off and vice-versa.

3.4.2. Puzzle inputs & outputs

The inputs and outputs of this puzzle have been taken from the design document of the previous group which worked on this project (??). This input and output diagram has been shown in Figure 10.

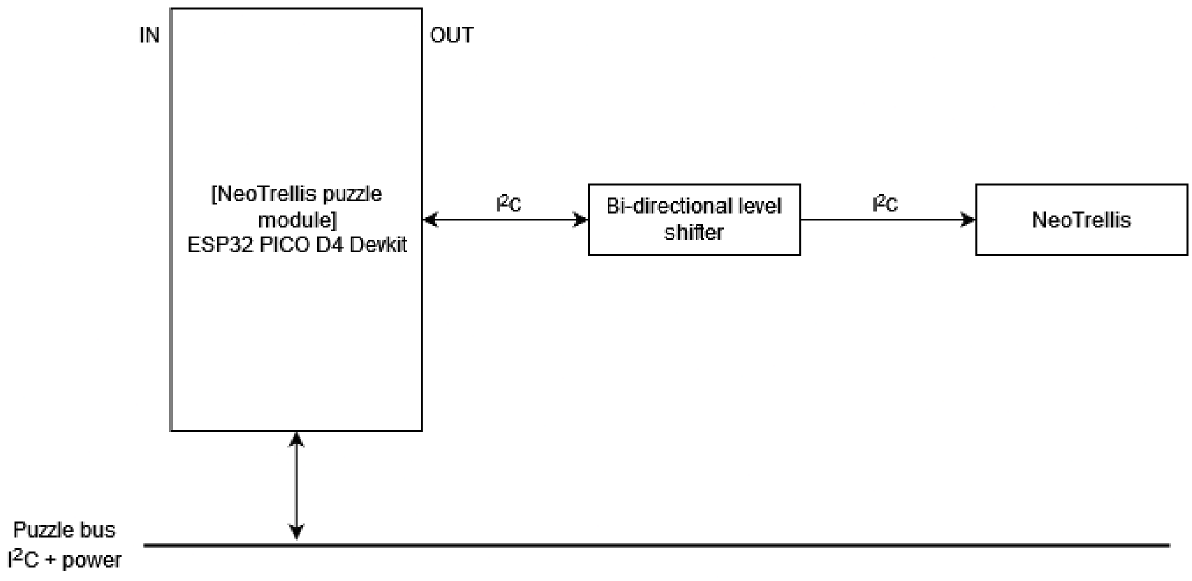

Figure 10. NeoTrellis puzzle in-out

3.5. Software puzzle

This subsection defines aspects of the 'software puzzle' module and gives a summary of how the puzzle is meant to be solved. This module will be created to facilitate the software puzzle game logic and communication with the main controller about the software puzzle state.

3.5.1. Software puzzle gameplay

The software puzzle consists of 12 input ports which can be connected using a banana plug connector. The 6 input ports on the left side of the puzzle each have their own logical circuit engraved in the box, and the 6 input ports on the right side of the puzzle have a letter (A through F) engraved in the box. The way to solve the puzzle is by connecting the banana plug cable from an input port on the left side of the puzzle to the corresponding input port on the right side of the puzzle.

When the puzzle starts, the participants of the game will have 6 code-fragments written on paper, corresponding to the logical circuits on the puzzle box. The bomb participants will have description of the C-code fragments, while the puzzle box participants only have the logical circuits on the puzzle box. The participants must communicate with each other to figure out which a fragment of C code corresponds with a logical circuit engraved on the puzzle box. Once this has been done the puzzle box participants can use a banana plug cable to connect the input and output to each other. Once the correct combination of logical gates with the correct letter is made, the puzzle is solved (shown by an LED lighting up above the puzzle). Allowing the participants to both see a binary code using 16 LEDs above the puzzle, and to continue to the next puzzle.

3.5.2. Puzzle inputs & outputs

As stated in Section 3.5.1 the puzzle has 12 inputs, as well as an LED which shows whether the puzzle has been solved and 16 LEDs showing a binary code. This is shown in Figure 11.

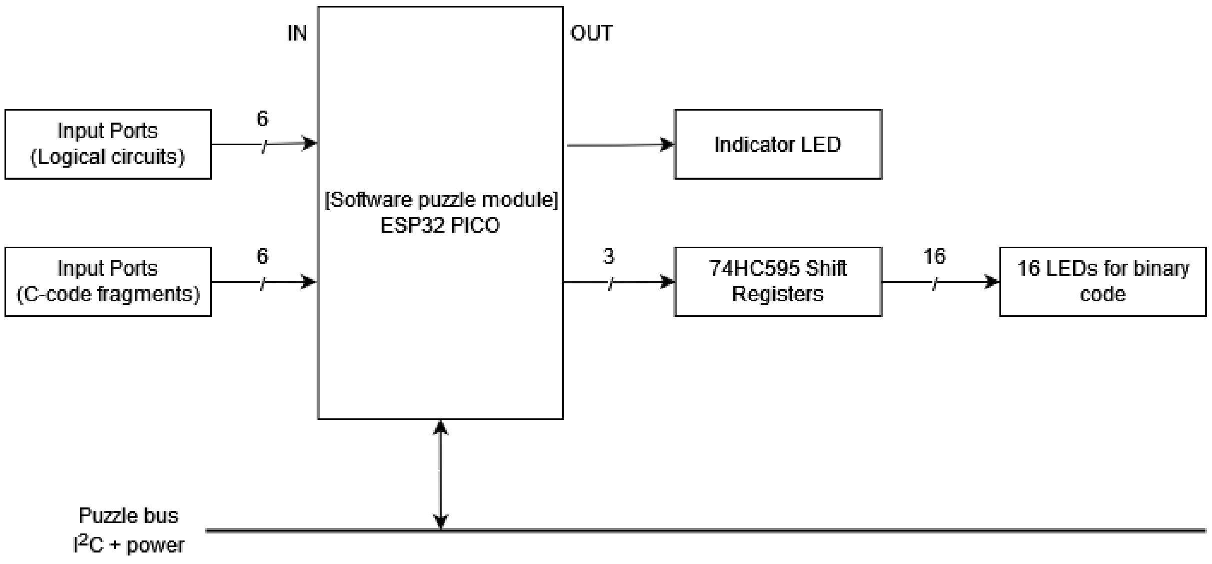
IN
OUT
Input Ports (Logical circuits)
6
Input Ports (C-code fragments)
6
[Software puzzle module] ESP32 PICO
Indicator LED
3
74HC595 Shift Registers
16
16 LEDs for binary code
Puzzle bus
I²C + power

Figure 11. Software puzzle in-out

3.6. Hardware puzzle

3.6.1. Hardware puzzle gameplay

The hardware puzzle has a logic gate puzzle engraved on the front plate of the puzzle box. To solve the puzzle, the user must set the toggle switches to the correct position. To solve the puzzle, a truth table is used.

The second part of the puzzle is unlocked after solving the logic gate puzzle, the user has to listen to a Morse code from which a four-digit code follows. The user then turns potentiometers to change this code on the display. The puzzle is solved when the user has put the correct code on the display. Once successful, the indicator LED will light up.

3.6.2. Puzzle inputs / outputs

The inputs and outputs of this puzzle have been taken from the design document of the previous group which worked on this project (21-22). This input and output diagram has been shown in Figure ??.

3.7. Vault puzzle

3.7.1. Vault puzzle gameplay

The vault puzzle is a puzzle created to test the communication skills of the student. It shows a code on the puzzle box, which then needs to be given to students with the game manual, who communicates this to the students at the puzzle box the button they must click. This needs to be done 5 times before the vault opens and the last code is given to defuse the bomb if a wrong button is clicked the vault resets and they need to start over from the beginning.

3.7.2. Puzzle inputs & outputs

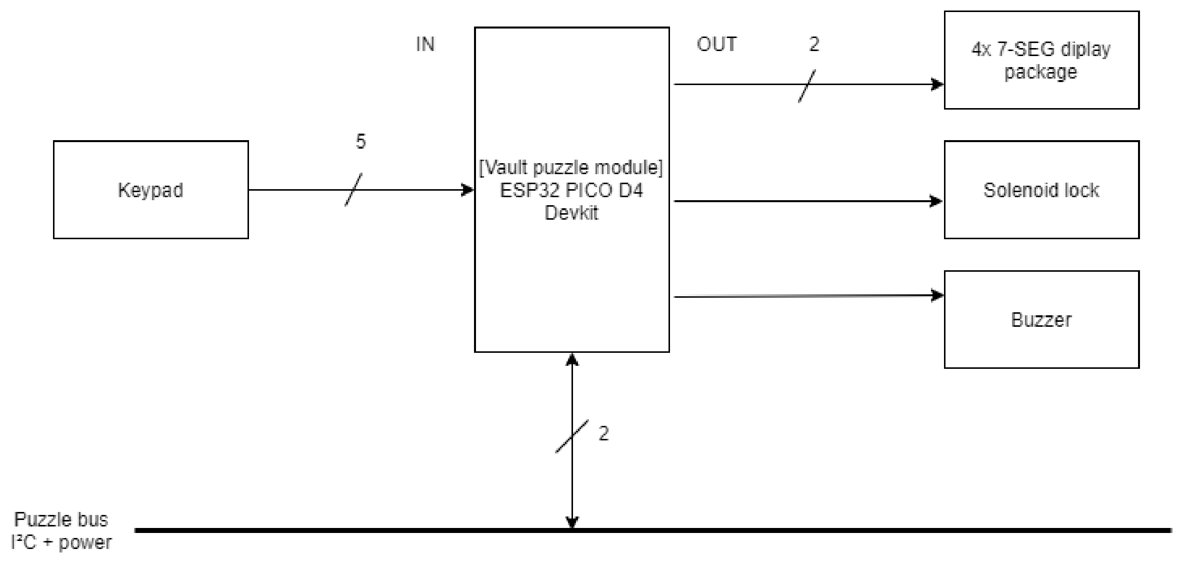

Figure 12. Vault puzzle in-out

Appendix A: References

[1] L. L. Blansch, E. Hammer, L. Faase, and T. in ’t Anker, “puzzlebox Doxygen documentation,” 2024. [Online]. Available: https://media.pipeframe.xyz/puzzlebox/23-24/doxygen.

[2] L. L. Blansch, E. Hammer, L. Faase, and T. in ’t Anker, “Research Document,” Avans University of Applied Sciences, 2024.

[3] L. L. Blansch, E. Hammer, L. Faase, and T. in ’t Anker, “Handover Report,” Avans University of Applied Sciences, 2024.

[[2122_handover]][4] L. van Gastel, J. de Bruin, T. Rockx, A. van Kuijk, and J. Baars, “Overdrachtsdocument,” Avans University of Applied Sciences, 2022.

[[2122_design]][5] L. van Gastel, J. de Bruin, T. Rockx, A. van Kuijk, and J. Baars, “Design document,” Avans University of Applied Sciences, 2022.

Appendix B: Glossary

RPI
Raspberry Pi

Main board
The main board is the PCB on the bottom of the puzzle box, this communicates with the puzzles and the bomb

Puzzle box hub
The puzzle box hub communicates with the puzzle box and the bomb, as well as helps with configuring them

SID
Security identifiers

game operator
Person who organizes a puzzle box play session

[1] This is not a typo