Low-Cost Control System for Pop-Up Escape Room

A Thesis submitted by

Mr Kieran Bryce Davey

[redacted]

For the award of

Bachelor of Engineering (Honours)(Computer Systems)

2024

University of Southern Queensland

School of Engineering

ENP4111 Research Project

Limitations of Use

The Council of the University of Southern Queensland, its School of Engineering, and the staff of the University of Southern Queensland, do not accept any responsibility for the truth, accuracy or completeness of material contained within or associated with this dissertation.

Persons using all or any part of this material do so at their own risk, and not at the risk of the Council of the University of Southern Queensland, its School of Engineering or the staff of the University of Southern Queensland.

This dissertation reports an educational exercise and has no purpose or validity beyond this exercise. The sole purpose of the course pair entitled “Research Project” is to contribute to the overall education within the student’s chosen degree program. This document, the associated hardware, software, drawings, and other material set out in the associated appendices should not be used for any other purpose: if they are so used, it is entirely at the risk of the user.

University of Southern Queensland
Faculty of Health, Engineering and Sciences
ENG4111/ENG4112 Research Project

Certification of Dissertation

I certify that the ideas, designs and experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged.

I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

K. Davey


ABSTRACT

Keywords: Escape Room, Configurable Embedded Control System, Master-Slave Architecture, JSON.

The escape room industry provides an effective medium for recreational and educational activities. Significant accessibility challenges exist in automating the narratives of escape room experiences for non-technical designers due to resourcing costs. This project addresses these challenges by designing, developing and evaluating the suitability of a low-cost control system for escape room usage. The research aims to implement flexible progression logic and wireless communication through configuration files to reduce the technical barriers to industry adoption.

The control system was developed as a scalable master-slave architecture using JSON files for system configuration and communication. Bluetooth BLE was employed to achieve low-cost, multi-room wireless communication, while nested JSON structure facilitates the representation of hybrid progression logic paths. The control system components underwent unit, integration and system testing to demonstrate the ability to meet industry-specific objectives and outcomes in controlled conditions.

The study contributes a solution to the escape room industry by addressing the technical barriers introducing significant resourcing costs. Integrating nested JSON configuration files into the master-slave control architecture provides an abstract interface for developing immersive escape room narratives. The system was validated within controlled conditions. Therefore, future work should address real-world operational tests and the development of additional tools, such as graphical web interfaces for configuration file compilation and real-time monitoring.

The work presented within the study provides a strong foundation for a low-cost control system for pop-up escape rooms. This makes complex escape room narratives more accessible for recreational and educational applications.

ACKNOWLEDGEMENTS

I want to thank my supervisor, Ms Catherine Hills. I acknowledge her continual support and wisdom throughout the year. She consistently guided the project's progress despite being very busy with her role at the University. I am thankful for her time, patience, and kindness, demonstrated by going above and beyond what was required. I am grateful towards the University of Southern Queensland for providing the facilities and content to learn the information necessary to complete this project.

I am deeply grateful for the support of Brenden Davey, Vicki Davey, Elijsha Krushka, Issac Davey, Samuel Davey, Samuel Krushka and Tram Le Kim. This is my family, and they have been load bearers when I struggled throughout the project. They consistently encouraged me to push through the year-long research project and to strive for a high standard of excellence. I am also thankful to my close friends Luke Whittingham, Isaac Humber, and Jacob Moffatt. Your friendship has provided support and encouragement.

I will never forget the sacrifice of Mrs Wendy Horne and Mr Allen Westman. As I struggled with dyslexia growing up, both mentors believed and turned my educational weaknesses into strengths.

I acknowledge the mighty hand of my saviour, Yeshua ha'Mashiah. I would not be here without my Elohim, who has been my rock, light, and sure foundation throughout this project.

I dedicate this project to all those listed above. Thank you for your belief and commitment, as this project wouldn't exist without you.

TABLE OF CONTENTS

ABSTRACT i
ACKNOWLEDGEMENTS ii
LIST OF TABLES viii
LIST OF FIGURES x
ABBREVIATIONS xi
CHAPTER 1: INTRODUCTION 1
1.1. Introduction and Background 1
1.2. Objectives and Aims 2
CHAPTER 2: LITERATURE REVIEW 4
2.1. Overview of the Escape Room Industry and Design 4
2.1.1. Escape Room Industry 4
2.1.2. Technologies within the Escape Room Industry 6
2.1.3. Escape Room Design and Implementation 7
2.2. Control System Architectures for Escape Rooms 8
2.2.1. Overview of Control System Architectures 8
2.2.2. Control System Requirements 9
2.3. Communication Strategies for Escape Room Control System 10
2.3.1. Overview of Wireless Communication Methods 10
2.3.2. Bluetooth Wireless Communication 11
2.3.3. LoRa Wireless Communication 12
2.3.4. Wi-SUN Wireless Communication 13
2.4. Configuration and Logic Definition Files 13
2.4.1. Overview of Configuration and Logic Definition Files 13
2.4.2. Configuration and Control Logic File Format Types 16
2.4.3. Abstract Data Types and Structures for Control Logic 18
2.4.4. Control System Logic Programming 19
2.5. Literature Review Summary 21
CHAPTER 3: METHODOLOGY 25
3.1.1. Research Design 25
3.1.2. List of Materials 27
3.1.3. Risk Assessment 27
3.1. Development Process 27

3.1.1. Embedded Architecture of Master Controller and Slave Controller. 28
3.1.2. Embedded JSON Serialiser 30
3.1.3. Bluetooth BLE Connection Interface 31
3.1.4. Master Controller Configuration 32
3.1.5. Slave Controller Configuration 33
3.1.6. Master Controller's Control Logic Engine 34
3.1.7. Slave Controller's Control Logic Engine 35
3.1.8. Command Message Communication 37
3.1.9. Master Controller's and Slave Controller's Communication Engine 38
3.1.10. Master Controller Initialisation and Slave Controller Initialisation. 39
3.1.11. Master Controller Configuration and Control Logic Engine Integration. 39
3.1.12. Master Controller's Control Logic Action Dispatcher and Communication Action Dispatch Integration 39
3.1.13. Slave Controller's Control Logic Action Dispatcher and Communication Action Dispatch Integration 40
3.1.14. Operational Escape Room Purpose Testing 40
3.1.15. Operational Escape Room User Purpose Testing 41
3.1.16. Operational Communication Coverage Performance Testing ... 42
3.1.17. Operational Communication Latency Performance Testing 42
3.1.18. Concurrent Slave Controller Communication to Master Controller Performance Testing 43
3.1.19. Concurrent Control Logic Evaluation Performance Testing 43
3.2. Testing Regime 44
3.2.1. Unit Testing 44
3.2.2. Integration Testing 79
3.2.3. System Testing 84
3.2.4. Performance Testing 87
CHAPTER 4: RESULTS 94
4.1. Testing Process 94
4.1. Embedded Architecture of Master Controller and Slave Controller 94

4.2. Embedded JSON Serialiser 100
4.3. Bluetooth BLE Connection Interface 107
4.4. Master Controller Configuration 109
4.5. Slave Controller Configuration 110
4.6. Master Controller's Control Logic Engine 111
4.7. Slave Controller's Control Logic Engine 120
4.8. Communication Engine 123
4.9. Master Controller and Slave Controller Communication Engine .. 127
4.10. Master Controller Initialisation and Slave Controller Initialisation . 128
4.11. Master Controller's Configuration Engine and Control Logic Engine Integration 129
4.12. Master Controller's Control Logic Engine and Communication Engine Integration 130
4.13. Slave Controller's Control Logic Engine and Communication Engine Integration 131
4.14. Operational Escape Room Purpose Testing 132
CHAPTER 5: DISCUSSION 136
5.1. Introduction to Discussion 136
5.1.1. Research Objectives and Outcomes 136
5.1.2. Overview of Key Findings 137
5.2. Interpretation of Results 138
5.3. Implications of Findings 139
5.3.1. Implications for educational and recreational accessibility. 139
5.3.2. Implications for control system scale and flexibility in escape rooms. 139
5.4. Limitations of Study 140
5.4.1. Overview of Limitations 140
5.4.2. Industry Specific Limitations 140
5.4.3. Limitations of Test Regime 141
5.4.4. Limitations of Design Decisions 141
5.4.5. Limitations of Constrained Scope 142
5.5. Suggestions for Future Research 142
CONCLUSION 143
REFERENCES 144

APPENDICES 148
6.1. Appendix A: Risk Assessment 148
6.2. Appendix B: Budget and List of Materials 149
6.3. Appendix C: Unit Tests 150
6.3.1. UT01 - Embedded Architecture of Master Controller 150
6.3.2. UT02 - Embedded Architecture of Slave Controller 156
6.3.3. UT03 – Embedded JSON Serialiser 164
6.3.4. Bluetooth BLE Connection Interface 184
6.3.5. Master Controller Configuration 185
6.3.6. Slave Controller Configuration 185
6.3.7. UT11 - Master Controller’s Control Logic Engine Parser 185
6.3.8. UT12 - Master Controller’s Control Logic Engine Interpreter .. 186
6.3.9. UT13 - Master Controller’s Control Logic Engine State Manager 186
6.3.10. UT14 - Slave Controller’s Control Logic Engine Action Dispatcher 187
6.3.11. Command Message Communication 187
6.4. Appendix D: Integration Testing 188
6.4.1. Master Controller and Slave Controller Communication Engine 188
6.4.2. Master Controller Initialisation and Slave Controller Initialisation 188
6.4.3. Master Controller’s Configuration Engine and Control Logic Engine Integration 188
6.4.4. Master Controller’s Control Logic Engine and Communication Engine Integration 188
6.4.5. Master Controller’s Configuration Engine and Control Logic Engine Integration 189
6.4.6. Slave Controller’s Control Logic Engine and Communication Engine Integration 189
6.5. Appendix E: System Testing 190
6.5.1. Operational Escape Room Purpose Testing 190
6.5.2. Operational Escape Room User Purpose Testing 190
6.6. Appendix F: Performance Testing 191

6.6.1. Operational Communication Coverage Performance Testing . 191
6.6.2. Operational Communication Latency Performance Testing 191
6.6.3. Concurrent Slave Controller Communication to Master Controller Performance Testing 191
6.6.4. Concurrent Control Logic Evaluation Performance Testing 191

LIST OF TABLES

Table 1: Unit Test UT01 44-45
Table 2: Unit Test UT02 45-46
Table 3: Unit Test UT03 46-47
Table 4: Unit Test UT04 47-49
Table 5: Unit Test UT05 49-50
Table 6: Unit Test UT07 50-52
Table 7: Unit Test UT08 52-53
Table 8: Unit Test UT10 53-55
Table 9: Unit Test UT11 55-57
Table 10: Unit Test UT12 57-59
Table 11: Unit Test UT13 59-61
Table 12: Unit Test UT14 61-63
Table 13: Unit Test UT15 63-66
Table 14: Unit Test UT16 66-68
Table 15: Unit Test UT17 69-70
Table 16: Unit Test UT18 71-73
Table 17: Unit Test UT19 73-74
Table 18: Unit Test UT20 74-75
Table 19: Unit Test UT21 75-77
Table 20: Unit Test UT23 77-78
Table 21: Unit Test UT24 78-79
Table 22: Integration Test IT03 79-80
Table 23: Integration Test IT04 80-81
Table 24: Integration Test IT05 81-82
Table 25: Integration Test IT06 82-83
Table 26: Integration Test IT07 83-84
Table 27: System Test ST01 85-86
Table 28: System Test ST02 86-87
Table 29: Performance Test PT01 87-88
Table 30: Performance Test PT02 89-90
Table 31: Performance Test PT03 90-92

Table 32: Performance Test PT04 92-93
Table 33: Master controller ESP32-C6-DevKit-1 v1.2 pin assignment 94
Table 34: Slave controller ESP32-C6-DevKit-1 v1.2 pin assignment 95
Table 35: UT01 outcome matrix 97
Table 36: UT03 Outcome Matrix 103
Table 37: UT04 Outcome Matrix 107-108
Table 38: UT05 outcome matrix 109
Table 39: UT07 outcome matrix 110
Table 40: UT08 outcome matrix 110
Table 41: UT10 outcome matrix 111
Table 42: UT11 test case outcomes 118
Table 43: UT12 test case outcomes 119
Table 44: UT13 outcome matrix 119
Table 45: UT13 outcome matrix 120
Table 46: UT15 outcome matrix 121
Table 47: UT16 outcome matrix 121
Table 48: UT18 outcome matrix 122
Table 49: UT19 outcome matrix 122-123
Table 50: UT19 outcome matrix 123-124
Table 51: UT20 outcome matrix 124-125
Table 52: UT21 outcome matrix 125-126
Table 53: UT21.2 outcome matrix 126
Table 54: UT23 outcome matrix 126
Table 55: UT24 outcome matrix 127
Table 56: IT03 outcome matrix 128
Table 57: IT04 outcome matrix 128
Table 58: IT05 outcome matrix 129
Table 59: IT06 outcome matrix 130
Table 60: IT07 outcome matrix 131-132
Table 61: ST01 outcome matrix 133-135

LIST OF FIGURES

Figure 1: Methodology Stage 25
Figure 2: Master Controller Architecture..... 28
Figure 3: Slave Device Architecture 29
Figure 4: peripheral_update process flow 99
Figure 5: json_schema_serialiser flow diagram 101
Figure 6: get_schema_content function prototype 102
Figure 7: SchemaPropertyMapping type definition 102
Figure 8: master control logic parser abstracted process diagram 112
Figure 9: traverse_expression process diagram 113
Figure 10: ExpressionNode relationship to sub-types 114
Figure 11: expression structure relationships 115
Figure 12: Logic expression containing structures 116
Figure 13: ST01 hybrid progression path 133

ABBREVIATIONS

API:	Application Peripheral Interface
FIFO:	First In, First Out
FPGA:	Field Programmable Gate Array
FSM:	Finite State Machine
HMI:	Human Machine Interface
JSON:	JavaScript Object Notation
LED:	Light Emitting Diode
MCU:	Microcontroller Unit
RPN:	Reverse Polish Notation
RSSI:	Received Signal Strength Indicator
SD card:	Secure Digital card
SPI:	Serial Peripheral Interface
XML:	Extensible Markup Language

CHAPTER 1: INTRODUCTION

1.1. Introduction and Background

In recent years, escape rooms have emerged as a popular form of entertainment and problem-solving and have been recognised for their educational value (Li et al., 2018). Increasing research has recommended integrating escape room activities into formal, vocational, and adult education due to their interactive experience (Staneva et al., 2023). Escape rooms require teams of people to coordinate their collective competency and problem-solving skills to solve puzzles to escape their trapped rooms. The design and creation of exciting escape room designs require various technologies, ranging from mechanical puzzles to complex digital interfaces. Startup escape rooms have leveraged multiple technologies and complex control systems to create these compelling experiences (Inés Tejado et al. 2021). The cost of setting up and operating an escape room can vary widely; however, engineering sophisticated escape room logic, communication systems and control architecture for the puzzles is a significant investment (Tercanli et al., 2021). The system architecture must also be re-designed and engineered whenever the escape room experience or story is rewritten, requiring recurring investment to stay competitive. The review of such challenges demonstrates that engineering escape room puzzles and systems presents significant hurdles, particularly in integrating physical and digital elements. Research has proposed that virtual reality-based environments for escape rooms can mitigate the high material and integration costs (Darejeh, 2023). However, a solution for reducing the technical challenges and budget limitations for physical escape rooms remains unsolved.

Despite the growing popularity of escape rooms, there exists a noticeable gap in the market for a scalable, cost-effective control system that can accommodate the rapidly changing storylines of the escape room industry. Current escape room control systems require technical knowledge or have a high financial cost to produce an immersive narrative (Tercanli et al. 2021). This project proposes an abstracted, low-cost control system architecture tailored for non-technical escape room owners. The proposed system aims to implement a control system that runs on modular, wireless embedded systems, allowing game progression logic to be customised without programming knowledge. The game progression logic will be defined within a game file data structure and can be inserted into the reusable master-slave architecture via

an SD card. The system is designed to allow non-technical escape room designers to define game progression rules and logic. The research will investigate a scalable data structure supporting complex logic designs. The study will also examine the game progression file's ability to be compiled from highly abstracted user interfaces and interpreted by reusable embedded systems.

The project seeks to reduce the technical challenges and high financial costs sophisticated escape room owners encounter when designing new experiences. The project will enable the design of escape rooms to become highly abstracted from the hardware level, potentially creating a drag-and-drop graphical design experience.

1.2. Objectives and Aims

Specific Objectives:

- The design of a low-cost master-slave control system architecture is suitable for escape room puzzles.
- Create a flexible escape room, progression logic file type and data structure that can be loaded onto the master and slave controllers via SD card.
- Implement master and slave interpreter for the game progression logic file data structure, ensuring accurate game progression and puzzle state management.
- Investigate coordination and scheduling schemes for master-slave communication, prioritising puzzle querying based on game progression.
- Analyse how different data file schemes impact system scalability and complexity.
- Evaluate and implement appropriate wireless communication methods, protocols and technologies for the master and slave controllers.
- Investigate system usability limitations. Analysing system latency, data rate, loss tolerance, wireless communication distance and response time.
- Evaluation of communication strategies such as suitability of polling versus interrupt-based methods for communication between master and slave devices.

If time permits,

- Develop a graphical web application to create the game progression logic.
- Develop a compiler that compiles the game progression logic into the master and slave game progression logic file.

Expected Outcomes:

- Low-cost, master controller embedded system with wireless communication, SD card reader and game file interpreter.
- Low-cost, slave controller embedded system capable of wireless communication, SD card reader, game file interpreter and peripheral API for puzzle control.
- A robust communication method and data protocol tailored for master-slave interactions within the escape room environment.
- Definition and implementation of a universal game progression logic file data structure and file type that supports complex escape room game design.
- A game progression logic file interpreter on both master and slave devices, ensuring correct puzzle state management, game progression and data transfer.
- Implementation of an effective coordination scheme in the master controller to manage and query slave devices based on the current stage of the game progression.
- Understand the usage limitations and scalability of the modular control system.

If time permits,

- A web application for designing escape room game logic.
- Error handling and input validation of web application design tool user input.
- Implementation of a compiler to generate required game file format for master and slave controllers from graphical web application representation.

CHAPTER 2: LITERATURE REVIEW

2.1. Overview of the Escape Room Industry and Design

2.1.1. Escape Room Industry

According to Gordon et al. (2019) findings, escape room activities positively influenced participants' perception of collaboration and teamwork compared to their perception of teamwork before the activities. Within the qualitative survey, 89% of students agreed that “I enjoy working in a team environment” after the escape room exercise as opposed to 79% before the exercise. 99% of the students strongly agreed that “I am an integral member of the team” after the exercise, as opposed to 94% before the escape room activities (Gordon et al., 2019). Escape room activities have also been shown to encourage problem-solving and cognitive function. The study conducted by Kinio et al. (2019) demonstrated that 75% of participants experienced a greater ability to retain the information from the interactive learning experience provided by the escape room. 92% of the students stated that the interactive learning format was appropriate for testing their knowledge retention (Kinio et al., 2019). The escape room activities undertaken by the students in both these studies suggest that escape rooms encourage not only effective cognitive function but also interactive social behaviour. Escape rooms are an effective medium for positively influencing the participants’ perception of teamwork while solving problems.

The role of escape rooms as an educational tool has been utilised to promote engagement within low-motivation course content. The gamification of such educational material has increased student motivation and promoted transfer learning across theoretical and practical skills (Sánchez-Martín et al., 2020). Sánchez-Martín et al. (2020) conveyed that escape rooms can allow multiple methods for collecting data related to student learning. Escape rooms enable the students to be assessed through direct observation, questionnaires, and discussion groups. The engagement within these sessions increased due to the participants' personal experiences with the activities. Escape rooms have significantly impacted the education industry, facilitating greater student engagement and promoting critical thinking towards the learning activity. Cain (2019) derived that 91% of classroom students were more engaged in critical thinking due to the problem-solving associated with escape room activities.

89% of the classroom students also indicated that they enjoyed the escape room activity more than traditional education methods (Cain, 2019).

The escape room industry is becoming more competitive, and new escape room business trends have been consistently growing since 2014 (Spira, 2023). Spira (2023) indicates that one of the biggest challenges facing the escape room industry is the operating costs associated with building and testing new game narratives. This is magnified by the prolonged time it takes to develop new experiences and ensure that quality experience is produced. Another article by Lakomkina (2023) confirms this challenge by stating that escape room businesses' most significant challenge is the high initial investment required for equipment and software. Lakomkina also states that maintenance costs are high due to updating the escape room design and puzzles (Lakomkina, 2023).

Educational escape rooms also face a similar challenge of funding educational experiences when developing immersive experiences (Chang, 2019). These articles indicate that as the industry becomes increasingly competitive, there’s a greater need to create new compelling experiences to keep consumer interest. The most significant hurdles in meeting this consumer expectation are the high initial cost and prolonged time needed to develop the technology and software integration. Tercanli et al. (2021) conducted an extensive study on the practical implementation of escape rooms in education. The research found that educator's technical and creative knowledge significantly inhibited the adoption of escape rooms in education.

Along with the technical hurdle in creating immersive educational experiences, the time and financial resources required to develop educational escape rooms are a significant challenge in integrating technologies into education. This highlights two common challenges, the first being the monetary and time cost of creating the technological integration of immersive escape room experiences. The second challenge is the technical knowledge required to develop and connect multiple technologies.

2.1.2. Technologies within the Escape Room Industry

As the escape room industry is an emerging field of education and entertainment, there's limited literature categorising it into distinctive formats. However, multiple technology platforms have been researched to determine the ease of integration and implementation.

Kiruthika et al. (2022) explored the implementation of virtual reality to mitigate the resourcing costs associated with physical escape room props. Virtual reality escape room experiences enabled participants to interact with virtual objects and augmented audio-visual experiences. The user would then interact with the escape room through virtual user interface actions. The virtual environment allows the participants to experience interactions that would typically be difficult or unrealistic to implement within physical escape rooms, such as teleportation. The escape room environment was developed using 3D modelling and game development software. VR simulation headsets are then utilised to place the participant within the escape room environment, with the interactions being controlled through VR controllers (Kiruthika et al., 2022).

A standard format for escape room implementation is physical escape rooms. The participants must solve a combination of physical puzzles to progress into the next room. The development of the escape room puzzles requires a minimum competency in electronic design, with the programming complexity depending on the progression logic of the narrative (Ross, 2019). Ross (2019) explored a low-cost escape room puzzle design that required the escape room designer to acquire the electronic components, assemble the electronic components, and then configure the Arduino microcontroller by modifying and uploading code. The puzzles were implemented using Arduino Nano, LCD, number pad, speaker, and batteries. This resulted in an interactive puzzle that gave feedback to the user through the LCD screen resulting from user keypad input. There are eight challenges when designing a physical escape room, these being balancing the difficulty of the puzzles for the user, engineering competency in designing puzzle logic, creating room elements that don't break easily, integrating new puzzles into the existing narrative, getting the timing right between puzzles, developing a reconfigurable narrative and playtesting the escape room when developing (Ross, 2019). These challenges highlight the difficulty and technical knowledge required to design and develop physical puzzles. It was determined that

advanced electronic and programming understanding would be essential for amending the system configuration. Ross (2019) found a gap in the system's ability to integrate with other puzzles and suggests integrating the puzzles with WIFI to configure and manage multiple puzzles simultaneously.

2.1.3. Escape Room Design and Implementation

The design of an escape room should contribute towards motivating an engaging learning experience. This motivation is achieved by containing the participants within a locked room and requiring them to solve puzzles to escape. Escape rooms are often designed as fictional locations, facilitating an engaging learning experience. The complexity of automated fictional scenarios complicates the design of escape room stories. The complexity is introduced as the escape room game designer must consider an enjoyable, immersive experience without detracting from the educational lessons (Elmet Project, 2021). According to Tercanli et al. (2021), the narrative progression of escape rooms can be open, sequential, or path-based. An open escape room design allows the participant to solve the puzzles in any order to escape. The participants interacting with an open progression path narrative must complete the puzzles within the narrative successfully. However, the order of completion does not impact the successful escape of the room. Sequential escape room progression requires solving the puzzles in a particular sequential order. One puzzle completion would lead the participant to the next, allowing for successful completion once all puzzles are successfully passed in order. The hybrid path-based progression logic combines both open and sequential progression. The progression logic of such an escape room would depend on the outcome of the previous and current stages of the escape room (Tercanli et al., 2021). Hybrid path-based progression logic allows the narrative to branch into different paths depending on the outcome of puzzle interactions.

When designing escape rooms, a balance between entertainment value, progression logic complexity, and educational value is needed. When developing the escape room experience, balancing these necessary components is complicated for non-technical designers. The escape room designer must determine the educational competency

objectives, construct an engaging narrative, and then program the narrative's progression path into the control system's software and hardware design.

Ross & Bennett (2022) identified a diverse range of physical escape room puzzles that can exist in an escape room environment. Four escape rooms were designed, each with three puzzles. The diversity and strategic placement of the puzzles within each escape room were designed to suit different player skillsets. This promotes team collaboration and ensures each player within the team contributes. The first escape room was based on digital electronics competency. The three puzzles included C decoding, waveform decoding, and 7-segment display understanding. The second escape room was based on electronic hardware competency and included a measuring voltage, continuity testing, and LED lighting puzzle. The third escape room targeted STEM activities for high school engagement around STEM disciplines. The third escape room included puzzles on hydraulics, rotational equilibrium, and Caesar cipher. The fourth escape room was designed for testing international tertiary students. The three puzzles in this escape room tested Australian slang, Australian Geography and Australian Inventions (Ross & Bennett, 2022). The escape rooms designed by Ross & Bennett (2022) demonstrated that puzzles can be designed to support a variety of educational domains. However, the designer needs to be competent in electronic and programming disciplines to implement the puzzles. Escape room control systems must also be extensively tested to ensure the solution is possible without committing to developing the progression logic (Ross & Bennett, 2022).

2.2. Control System Architectures for Escape Rooms

2.2.1. Overview of Control System Architectures

Designing complex control systems is a technical and challenging process, even for competent escape room designers. The control system's design involves selecting controller hardware, interfacing the controller to the peripherals, and programming the control algorithms specific to the hardware implementation (Shaik, 2011). Shaik (2011) explored the implementation of a control system that utilised a 32-bit RISC microcontroller that offered compatibility with control system peripherals for real-time data acquisition. The study found that the microcontrollers' ability to utilize standard communication protocols allowed for network control and data acquisition from remote

locations. This emphasises the embedded system's ability to be scalable and flexible to expanding control systems and can be reprogrammed to meet various application requirements (Shaik, 2011). Peng et al. (2008) also investigated the approach to developing low-cost control systems by developing embedded controllers. The study found embedded control system architectures support advanced control logic through high computational capacity and open-source software packages. The microcontrollers' ability to communicate directly with sensors and actuators allows for a simplified development approach yet can also be scaled beyond the microcontroller's pinout through network capabilities (Peng et al., 2008).

Hanou et al. (2020) implemented a control system and interface for monitoring and configuring an escape room using the client-server architecture. The control system comprises a back-end server, a front-end user interface, and client computers. The back-end server was implemented using a Raspberry Pi, which managed the message broker, serving web interface and client device management. The back-end server serves the front-end and allows the escape room employee to monitor the state of the escape room. The client devices are puzzles that all have a unique IP address. The IP address allows client-server communication, which passes escape room state updates (Hanou et al., 2020). The control system implemented by Hanou et al. (2020) successfully passed field testing with suitable performance metrics. The control system requires all client devices and the server to have a unique IP address over the local area network and can connect to the network over WIFI.

2.2.2. Control System Requirements

The design and selection of a control system architecture needs to consider the functional requirements and real-time characteristics of the control process. Therefore, control systems for the escape room industry need to scale and meet the operational requirements of escape room design.

Ross and Bennett (2022) demonstrated that an educational escape room narrative can have multiple puzzles included within its storyline. This outlines a one-to-many relationship between the controller and the puzzles it will need to manage. Therefore, a requirement of the control system architecture is that its performance does not degrade as the number of puzzles being managed scales. Different puzzles within the

same escape room narrative can also require different datatypes to be processed by the controller to trigger completion stages (Ross & Bennett, 2022). This highlights the requirement for the controller to be able to process multiple data types within the same escape room narrative.

Ross (2019) demonstrated that each puzzle within an escape room narrative can have multiple sensors and actuators, which need real-time processing in response to user interaction. From this another control system requirement is that the controller can process many peripheral state values for each puzzle. As the state of the puzzle peripherals is updated by user interaction, another requirement is that data can be processed in real time by interrupt (Ross, 2019). Tercanli et al. (2021) outlined that the escape room narrative's progression logic can be sequential and open. This results in path-based combinational logic, requiring the control system to process multiple puzzle data packets concurrently (Tercanli et al., 2021).

2.3. Communication Strategies for Escape Room Control System

Understanding available wireless communication technologies and their typical characteristics is important for selecting a suitable option to achieve the control system's outcomes and objectives. The wireless communication methods assessed are Bluetooth, LoRa, and WiSUN. This contributes to achieving the objective of evaluating and implementing appropriate wireless communication methods, protocols, and technologies for the master and slave controller.

2.3.1. Overview of Wireless Communication Methods

Goncalves et al. (2021) evaluated three low-power wireless communication technologies for SmartGrid networking applications. SmartGrid networks provide real-time monitoring, control signals, and data transfer for multiple power system devices. This has a similar relational multiplicity as the requirements defined for the escape room control system. Wireless communication technology within the escape room control system will need to allow for real-time data processing for multiple devices. Goncalves et al. (2021) investigated the performance characteristics of Bluetooth, LoRa and Wi-SUN wireless technologies to assess their suitability for SmartGrid networks. Coverage, data rate, power consumption, interoperability, physical layer

complexity, topology and worldwide acceptance were evaluated in the assessment (Goncalves et al., 2021).

2.3.2. Bluetooth Wireless Communication

Park & Umirov (2012) presented the implementation use case of three different link-type profiles for Bluetooth communication in networked control systems. Serial Port Profile (SPP), Human Interface Device (HID) and Synchronous Connection-Oriented (SCO) were compared to assess their performance for communication between sensors, actuators, and controllers. SPP emulates a serial port over Bluetooth connection, allowing devices to communicate and simplify configuration and connection between devices. SPP was found to introduce non-linearity and unpredictable data packet ground, which can lead to latency issues (Park & Umirov, 2012). SCO is often utilised for real-time audio transmission using Bluetooth, as it allows for data streaming at low latency. The main limitation found with SCO is the limited support it provides in most Bluetooth modules. HID is commonly used for devices such as keyboards, mice and gaming controllers and is optimised for low-latency communication. HID is supported by a wide range of devices and operating systems, making it suitable for networked control systems (Park & Umirov, 2012). The position control of a DC motor utilising these different link type profiles found that SPP is not recommended in use cases where low latency is required; however, it allows for simple configuration with the host controller interface (HCI). HID is preferred for low-latency control systems; however, establishing connections with the HCI is more complex. This makes HCI more challenging in control systems that require direct control; however, it is the best choice where low latency and reliable connection are critical systems (Park & Umirov, 2012).

Goncalves et al. (2021) evaluated the performance characteristics of Bluetooth for SmartGrid systems with the following specifications. The coverage of Bluetooth was found to be limited to 100 meters, with the devices being assessed using a printed antenna of -6dBi gain. The coverage depended on how dense the obstacle conditions were within the operating environment. When operating within densely obstructed environments, the range was found to be 43m, while free space allowed for communication coverage of 242m (Goncalves et al., 2021). The data rate for Bluetooth is dependent on the version being implemented. Bluetooth 5.0 was found to allow data

rates of 1Mbps to 2Mbps. Bluetooth Low Energy provided data rates of 125kbps but could be increased to 500kbps when increasing the receiver sensitivity to -106.7 dBm. Bluetooth is suitable for battery-powered devices as it has low power consumption but depends on the chipset and other operating factors (Goncalves et al., 2021). Bluetooth allows for various network topologies, such as point-to-point connections and star configuration. This allows Bluetooth to be widely utilised globally, particularly in consumer electronics (Goncalves et al., 2021).

2.3.3. LoRa Wireless Communication

Long Range (LoRa) communication is suitable for physically mobile applications and establishing a private network without a communication provider. It can transfer small data packets over a long-range network, connecting up to 1 million nodes (Anani et al., 2019). A LoRa network consists of a gateway, network server, application server and nodes. All nodes communicate through the gateway and commonly utilise the star topology. Angelov et al. (2023) investigated the suitability of a narrowband LoRa communication network for managing and monitoring an IoT lighting system. The control system implemented the LoRa system with sensors and actuators serving as nodes that communicate with a central LoRa gateway. The LoRa gateway then collected the data in a cloud-based server through a standard Wi-Fi network (Angelov et al., 2023). The LoRa network utilised three different node modules, with the mini module having a maximum range of 500 meters and the standard node modules having a maximum range of 900 meters. LoRa was a suitable selection for reliable control system communication while implementing optimised configurations, which significantly improved communication distance and reliability (Angelov et al., 2023).

Goncalves et al. (2021) evaluated the performance characteristics of LoRa for SmartGrid systems. The coverage of LoRa was able to establish a connection from 10km with an external antenna of 2dBi gain. In densely obstructed areas, the coverage was found to be 1.9km. The data rate for LoRa devices ranged from 0.3kbps to 50kbps. LoRa was determined to be suitable for battery-operated devices as it is designed for low power consumption. The devices utilising LoRa are standardised through Semtech Corp, which developed the technique and manufactured these modules. LoRa is designed to be implemented with the star topology with a centralised gateway managing the node devices (Goncalves et al., 2021).

2.3.4. Wi-SUN Wireless Communication

The Wireless Smart Ubiquitous Network (Wi-SUN) is suitable for medium-range communication that requires low power consumption and high node density. Wi-SUN enables mesh topology, which can be complicated to configure but allows redundancy through network hopping if different nodes fail (Anani et al., 2019). Wi-SUN is suitable for metering infrastructure, distributed automation, and home area networks. The challenges related to Wi-SUN communication come from higher device costs and its tendency to be prone to interference (Anani et al., 2019). Kashiwagi et al. (2022) evaluated the suitability of Wi-SUN networks for the transmission performance of USB-type radio boards. The star and tree topology were tested, giving a packet transmission success rate over 95%. Due to multi-hop processing, the tree topology had a longer configuration time than the star topology (Kashiwagi et al., 2022). Both network topologies remained stable with no drop states for 12 hours of continuous operation. During this time, the power consumption was evaluated to enable the system to operate on two AA batteries for at least one year (Kashiwagi et al., 2022).

Goncalves et al. (2021) evaluated the performance characteristics of Wi-SUN for SmartGrid systems. Wi-SUN coverage established a connection from 7km with an external antenna of 2dBi gain. In densely obstructed areas, the coverage was found to be 1.3km. The data rate for Wi-SUN devices ranged from 50kbps to 300kbps. Wi-SUN was determined to be suitable for battery-operated devices as it is designed for low power consumption. Wi-SUN is designed to be implemented with the mesh topology, enabling it to scale well with the control system. The mesh topology does require more complicated communication management and can also have increased latency due to multiple network hops (Goncalves et al., 2021).

2.4. Configuration and Logic Definition Files

2.4.1. Overview of Configuration and Logic Definition Files

Configuration management allows for consistent operation of an embedded system and software across multiple product-level changes. Configuration management is a process and solution for system design that maintains the integrity of the system as it changes (TARAMAA et al., 1996). This is relevant to the implementation of an adaptable escape room control system as the same control system will need to be changed frequently to accommodate different escape room narratives. Suitable

configuration management will enable the embedded control system to evolve between narrative versions using a common configuration definition.

An industry-wide concern is that organisations have valuable software solutions that have consisted of different methodologies and technology stacks over time. Effective software configuration management allows organisations to adopt innovative solutions to remain competitive by defining consistent configuration schemes (TARAMAA et al., 1996). Mature configuration management solutions must implement change management controls and version management and are not dependent on specific hardware or software modules. Flexible software process design requires that configuration management and software process requirements are not codependent yet still retain the internal relationships between them. Configuration management solutions should facilitate the change management between existing software processes as they evolve (TARAMAA et al., 1996). TARA MAA et al. (1996) outline the process of defining configuration management into the stages of configuration identification, configuration control, configuration status accounting and configuration audit. The process of configuration identification involves defining the items of a product that will need configuration management. Configuration control defines the process and structures that support changes to the configuration items identified throughout the product life cycle. Configuration status accounting is the schemas that log and report the status of configuration items and their change requests. Configuration audit is the process that verifies the completion and correct implementation of the configuration items after changes (TARAMAA et al., 1996).

Configuration files allow developers to change the key-value pairs within an XML document to change the program settings, objects, and protected references without recompiling the system's source code (Kasbe, 2015). Effective implementation of configuration files enables flexibility in utilising the same source code across different system environments. Configuration files utilise XML key-value pairs to map system resources to reduce the cost of redeveloping source code for different control branches and initialisation cases. Kasbe (2015) focuses on the configuration files for .NET technology and outlines three different configuration file types. The application configuration file contains the pre-application configurations. The pre-application key-value pairs contain version control variables, enabling the system administrator to

select the versioning of source code modules, data storage paths and other initialisation values. The machine configuration file stores the global configuration values to be initialised across individual directories. The web configuration files store the configuration values for the web applications separate from the application source code configuration (Kasbe, 2015). Each of these three files is hierarchical, enabling configuration files further up the chain to overwrite the more granular values (Kasbe, 2015).

Gutjahr and Heumesser (2014) present a method for generating configuration files to maintain and administer computer systems. The configuration files generated define technical information and operational values to be monitored for a central server and its agents—the technical information defines device operational parameters such as allocated bandwidth or allowed communication protocols. The configured device can then send alerts and messages to the central server depending on the operational value thresholds defined (Gutjahr & Heumesser, 2014). Generating configuration files requires the central server to generate unique configuration file values for each agent. The generated configuration file must also generate the correct file structure and format depending on the agent architecture (Gutjahr & Heumesser, 2014). The method for generating configuration files by Gutjahr and Heumesser (2014) utilises a template XML file, XML data file and two XSTL style sheet sheets. The configuration file goes through three transforms to create the executable XML configuration file during generation. The first XSTL stylesheet transforms the XML data file where each query of the set of queries corresponds to an XPath expression (Gutjahr & Heumesser, 2014). The second XSTL, comprising the subset of parameter settings, takes the first transform and maps to the corresponding XPath expressions. The second XSTL maps the location of each parameter set to a location within the XML template (Gutjahr & Heumesser, 2014). The output from the second transformation is the XML configuration file, which is converted into its executable format.

Once the configuration files have been generated and validated, they must be distributed across the control system architecture. Lee et al. (2014) outlines a method that allows a master device to share its configuration files with its slave devices. This enables a control system to load a master configuration file to the master devices. Then, the master device distributes the corresponding configuration to each slave

device using the CANopen protocol (Lee et al., 2014). This enables the repair or update of specific slave devices within a control system from the master controller. The proposed implementation has the master device check the version of the configuration file against a directory on the host computer during system boot or startup. If the version stored on the master devices does not match the version on the host computer, the master device downloads the latest version from the host computer. The master devices then distribute the configuration object files to the slave devices in a feedback mode and validate that reconfiguration was successful. On successful reconfiguration, the master and slave devices may reboot depending on configuration settings (Lee et al., 2014).

2.4.2. Configuration and Control Logic File Format Types

Configuration management implementation across distributed systems can involve configuration files using different file types. This introduces the challenge of maintaining or generating files for a control system as the system evolves. Elsner et al. (2011) propose a framework that validates consistent models across multiple configuration format types, fixes the errors according to rules and serialises back into the original format type. The proposed framework investigates the compatibility between Ecore DSMs, XText DSLs, XML schema XML, Java Property files and C header files (Elsner et al., 2011). Each config file has its own model. The round-trip mechanism then converts the configuration file to its defined model using its metamodel, which maps the configuration artefacts to the model. The universal model is then validated and fixed if necessary. Once validated and fixed according to the metamodel constraints, the round-trip mechanism converts the universal model back into the original configuration file format (Elsner et al., 2011).

Chrysalidis and Frank (2024) implemented a universal configuration format that managed unsynchronised and decentralised data for avionic systems. The configuration format leveraged the universal model approach to manage the configuration changes. The configuration management solution identified the configuration data into three different groups: devices, testing and network. The configuration file generation process was based on the eclipse modelling framework. Then the meta-model definitions were converted into the custom Universal

Configuration Format for Avionics (UCoF) configuration format (Chrysalidis & Frank, 2024). This highlights the possibility of generating custom configuration formats based on meta-model definitions for application-specific use cases.

An alternative file format for configuration files is the JavaScript Object Notation (JSON) file format. The JSON file format can represent both the configuration of control system devices and the format that structures data for communication between control system devices (Wehner et al., 2014). This enables a single file format for configuration management and message parsing throughout the control architecture. JSON is an international data processing standard that is human-readable, data inter-changeable and lightweight file size format (Wehner et al., 2014). Wehner et al. (2014) experimented with a concept that utilised JSON to dynamically distribute the computational load of services across multiple FPGA nodes on an IoT network. The concept allowed users to stream video footage from one system service to another using the JSON file format. However, when the system receiving the streamed footage utilises all its resources, the image processing can be delegated to other nodes within the IoT network. This was achieved by utilising a standard JSON structure, allowing the streaming service to select the appropriate service by configuring key-value pairs. The payload of the image and its properties were also streamed within the same JSON data structure, enabling the image to be processed by the delegated node (Wehner et al., 2014). This demonstrates JSON's ability to configure control system devices while parsing the relevant data in a structured format.

Kasbe (2015) defines the implementation of configuration files as being in the Extensible Markup Language (XML) file format. XML stores the key-value pairs of object definitions through semantic tags (Kasbe, 2015). Despite being a standard format type for configuration files, XML has been proven to have greater processing overhead than JSON. This is due to the syntax structure of XML being more complex, requiring higher computation to parse. This can drain a significant proportion of the resources in embedded systems (Kasbe, 2015). This highlights the need for carefully selecting the configuration and data structure file type carefully depending on system resources and use case.

2.4.3. Abstract Data Types and Structures for Control Logic

Organised and universal data structures allow for scalable and descriptive representation of a control systems state. The design of a control system requires the key system objectives to be identified and represented within a universal data structure. Mapping the key system objectives within a universal data structure allows it to be de-structured by different sub-systems (Vojir & Beran, 2015). Developing a hierarchical composite structure allows for a clean organisation of system parameters and commands. This allows programmers to easily modify and navigate the data structures (Vojir & Beran, 2015). Another advantage of universal data structures is that they support modular design principles. This allows the data structure to scale and adapt the structure for different applications while keeping consistent model schemas. This enables the designed data structure to be compatible across all devices in the control system despite different hardware and internal processes (Vojir & Beran, 2015).

Fuzzy logic represents control logic within control systems, which requires tolerating imprecise data and modelling non-linear functions. Fuzzy logic can manage partial truth values instead of standard discrete Boolean values (Chrysalidis & Frank, 2024). This is achieved by allowing a degree of truthfulness or falsehood around the control variables, which closely mimics human thinking and decision-making. Fuzzy logic implementations allow abstract or complex problem-solving to be modelled within an expert system model, allowing for flexible decision-making (Chrysalidis & Frank, 2024). Chrysalidis and Frank (2024) outline that fuzzy logic is common within embedded control systems and can be found within vehicle sub-systems, air conditioners, digital image processing and pattern recognition applications. The main limitation of fuzzy logic systems is the complexity of designing the membership functions and rule bases, which describe the control problem. Fuzzy logic implementations can also lack precision depending on how fine-tuned the rule description is. This requires precision tolerance to be understood in order to test the control system (Chrysalidis & Frank, 2024). Bashi (2024) investigated the application of fuzzy logic to manage traffic flow at intersections. The system adjusted the signal timing based on real-time traffic conditions. This aimed to reduce waiting time at the intersection and improve traffic flow in different traffic densities (Bashi, 2024). The fuzzy logic controller ingested real-time traffic data and utilised image processing to

enhance the traffic condition parameters. The fuzzy logic controller then interpreted the traffic conditions based on the scale of each parameter to determine the wait time for each path (Bashi, 2024). Bashi (2024) found that the simulation reduced congestion and decreased vehicle wait time. The system also prioritised emergency vehicle paths, ensuring they experienced minimal delays (Bashi, 2024).

Finite State Machine (FSM) are real-time control structures that produce abstracted models from input alphabets. The input alphabet is converted into internal variables and states, producing an output alphabet that can be parsed as output values (Miroshnyk et al., 2018). FSMs are often represented as state diagrams, which visually describe the states, transitions between states and the actions of the FSM (Miroshnyk et al., 2018). Miroshnyk et al. (2019) developed a pattern for describing FSM in hardware description language for VHDL in FPGA applications. The resulting method utilised temporal state diagrams to represent a three-process control pattern. The temporal state diagram incorporates delays into the state diagram, enabling real-time control Miroshnyk et al., (2019). The results showed that the two-block FSM structure was successfully simulated and synthesised into the FPGA control system. This demonstrates FSM's ability to represent real-time logic control systems on embedded devices.

2.4.4. Control System Logic Programming

Many real-time control logic programming languages have been established, each with its own strengths and limitations. Some of these control structure languages are finite state machines (FSM), design structure diagrams (DSD), function block diagrams, ladder logic and sequential function charts (Mallaband, 1991). Mallaband (1991) established criteria for selecting the programming technique for real-time control systems. The main factors that need to be considered are the characteristics of the controlled system, the application domain of the control system, the familiarity with user training and experience, the architecture model that defines the control system and the features that need to be described by the programming technique (Mallaband, 1991).

Ladder logic diagrams have been a long-standing standard for representing control logic in programmable logic controllers. It has been the preferred language and widely accepted due to its fundamental programming elements mimicking discrete logic primitives (Wareham, 1988). Wareham (1988) identifies that a limitation of ladder logic is that it does not easily represent multiple events occurring concurrently. Rather, ladder logic is formatted so that the program scans the rungs of the control process sequentially (Wareham, 1988). Control rungs should be organised into zones and incorporate jump statements and sub-routines to facilitate the simultaneous processing of independent operations. This can make the program lengthy and difficult to design, so it scales with added complexity (Wareham, 1988).

An alternative to ladder logic is sequential function charts. Sequential function charts are an international standard representing the control logic graphically using function blocks, steps, and conditional transitions (Wareham, 1988). Sequential function charts can represent command- and event-driven systems and handle concurrency in a single structure (Mallaband, 1991). Mallaband (1991) outlined the specification for representing real-time control systems with sequential function charts. Function blocks represent the actions executed when a step is active. A functional block can either be stored or not, impacting whether the state persists beyond the step's activity. The steps represent specific states within the control process. The steps organise the control logic into distinct phases associated with control commands. Finally, conditional transitions define the conditions in which the control process moves between steps. The conditional transitions control the process flow based on the conditions defined (Mallaband, 1991).

Ivanescu et al. (2007) present a method for process control in embedded systems using sequential function charts. The method successfully demonstrated the ability of microcontrollers to interpret and execute multiple SFCs with efficient control processes. The major limitation of running SFCs on microcontroller devices is the memory capacity of embedded systems and the implementation requiring a reduced number of input/output pins (Ivanescu et al., 2007). Ivanescu et al. (2007) implemented this with an infinite loop function, which is continuously called three functions: input acquisition, SFC executions, and output update. The microcontroller interpreted each SFC step as a C function and executed the defined actions based on

the transition conditions. Once the step definition was interpreted, the microcontroller would execute the associated actions as one-time or continuous. The transition conditions were evaluated within each step function execution to determine if the next step was in the state (Ivanescu et al., 2007). The implementation also managed parallelism and convergence by separating converging elements into separate SFC charts. The main loop process would update pointers to ensure they were executed in the next cycle (Ivanescu et al., 2007). This demonstrates the practical application of SFC diagrams in embedded systems and their ability to handle complex control processes.

2.5. Literature Review Summary

The current literature review highlights the clear benefits of escape room experiences within the recreational and educational industries. The exercises promote team collaboration and social interaction through problem-solving (Gordon et al., 2019). The literature also demonstrates that escape room experiences enable an immersive learning experience and enhance transfer learning for low-motivation topics (Sánchez-Martín et al., 2020). With these benefits, the escape room industry continues to grow; however, it faces consistent resourcing challenges (Lakomkina, 2023). The resourcing challenges result from integrating the control technology with the quickly changing story narratives of the escape room design (Tercanli et al., 2021). This identifies a gap within the literature that needs to be resolved to allow non-technical escape room designers to develop escape room experiences at a low cost.

Multiple technologies have been researched to develop escape room narratives without the need for computer programming or electronic competency. Virtual reality escape rooms enable the users to interact within virtually immersive environments and allow the participants to interact with them through VR controllers and headsets (Kiruthika et al., 2022). Physical escape room experiences were shown to be able to test educational competency across a variety of educational domains but require the knowledge to configure and engineer the puzzles (Kiruthika et al., 2022). Ross (2019) outlines the need for the puzzles within the narrative to communicate with one another and integrate within a control system. This would enable the automatic progression throughout the escape room stages (Ross, 2019).

The control system architecture of an escape room would need to scale with a one-to-many relationship between the controller and its puzzles (Ross & Bennett, 2022). The puzzles within the control system would also require the controller to interpret different data types depending on user interaction with the puzzle. This highlights the need for the control architecture to process multiple datatypes within the same escape room narrative (Ross & Bennett, 2022). The control system would need to process a single puzzle's many sensor and actuator values. This would require that the control system be able to process multiple asynchronous puzzle interactions simultaneously. However, each puzzle interaction would also require processing many sensor and actuator values within each puzzle data structure (Ross, 2019). The control system logic representation would be required to represent path-based combinational logic (Tercanli et al., 2021). This control logic representation would need to scale with the many peripherals and puzzles that impact the system state. The path-based combinational logic presents the need for multiple data packets to be processed concurrently (Tercanli et al., 2021).

The performance characteristics of different wireless communication methods have been researched to achieve a pop-up escape room at a low cost. Bluetooth, LoRa, and Wi-SUN protocols have different performance characteristics and compatible topologies. Bluetooth was suitable for point-to-point and star topology communication use cases compatible with the master-slave architecture (Goncalves et al., 2021). Goncalves et al. (2021) state that Bluetooth can achieve a coverage of 43m in densely obstructed areas and provides data rates of 125kbps. There is a gap in understanding of the ideal implementation of wireless communication methods for low-cost escape room control systems. The selection of wireless communication methods will depend upon the performance characteristics and ease of automatic connection configuration.

Scalable configuration management would be required to implement a universal control system for differing escape room narratives. Well-defined configuration management would enable consistent control system operation independent of the hardware and process control logic (TARAMAA et al., 1996). Flexible implementation of control processes and hardware systems requires configuration management and systems to be not co-dependent. Rather, the relationship models between the system

and its configuration management definitions should be defined (TARAMAA et al., 1996). This would allow the escape room puzzles and controllers to structure their logic control and data structures so that evolving the narrative would not disrupt the system's operation. The controller source code would only need to be complied with once, and the configuration file would point to the updated data structures or control logic file. XML is the common file type utilized for configuration files (Kasbe, 2015). XML can represent the configuration variables within the file as key-value pairs. The JSON file format can represent the configuration file format and communication data structure with reduced processing overhead (Wehner et al., 2014). This would provide a lightweight file format for the puzzles and controller within the escape room to be configured and communicated. The requirements of the escape room control system will require a data structure format to represent both payload data and control logic. This was demonstrated by Wehner et al. (2014), who utilised JSON file format to transfer real-time video capture to processing nodes over the Internet of Things network. The data structure configured the target node by selecting complied services and parse the image payload for the process.

The abstract data types and structures that represent the control logic and payload will need to scale with the complex narrative of the escape room. Fuzzy logic is a paradigm that allows a model to fit non-linear control problems based on variable input values (Chrysalidis & Frank, 2024). Fuzzy logic control logic requires significant development time and expert advice to define the membership functions and rules. It can be very imprecise without a comprehensive set of rules for the control system. This is not ideal for escape room control systems as the state transitions between control stages can be defined by definitive goals. Finite state machines are real-time control structures that can be represented by abstracted models (Miroshnyk et al., 2018). The model definitions describe the transformation of input values and output values into modelled system states. This allows the control logic to be represented by state diagrams with clearly defined structure models (Miroshnyk et al., 2018). Sequential function charts map the state transitions of control system logic graphically. They allow for the scalable and concurrent handling of control processes that meet the requirements of the escape room control system (Mallaband, 1991). Since they are a graphical representation of control logic, SFC would be easier for non-technical escape room designers to understand.

There is a clear gap in the literature to represent complex escape room progression logic within a universal control architecture. Implementing a graphical programming diagram, such as sequential function diagrams on a low-cost embedded system, would reduce the technical understanding and financial barriers challenging the escape room industry. Developing a configuration management method would allow control progression logic to be separated from the compiled source code, which interprets and executes it. An escape room control system that reduces these barriers would provide greater access to immersive learning activities' educational and recreational benefits.

CHAPTER 3: METHODOLOGY

3.1.1. Research Design

The objectives and aims of the research seek to evaluate the specific implementation of a control system for the escape room industry. The research philosophy of pragmatism will guide its process. A pragmatic evaluation involves implementing the control system to determine its suitability for the escape room industry through experimental performance and functional case study tests. This requires a mixed-methods research approach to reach a comprehensive conclusion.

The quantitative evaluation will be designed as experimental tests to derive the performance characteristics of the control system. The performance tests will determine dependent values that are key metrics for successful escape room operation. The performance testing will be conducted during the Performance Evaluation stage shown in Figure 1: Methodology Stages. The performance metrics for evaluating the designed control system are operational wireless communication coverage, operational wireless communication latency, concurrent slave-to-master communication limitation and concurrent control logic variable evaluation limitation. The research will employ purpose sampling by collecting measurement data during controlled operating conditions. The sample data collection method is outlined in the Performance Testing section. The data obtained will then be analysed using ANOVA and regression analysis to evaluate how the control system performance scales as escape room complexity and locality variables change.

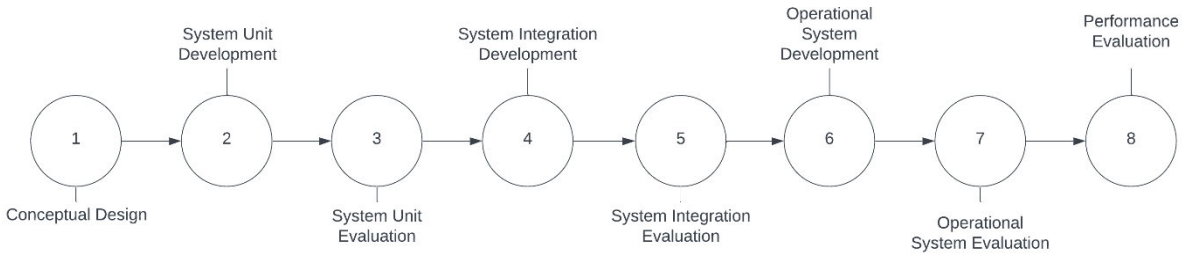


Figure 1: Methodology Stages

The qualitative evaluations will be designed as case study tests conducted at milestone stages of the development process. The qualitative evaluations will determine if the control system functions correctly within an operational escape room environment. Such qualitative tests will be conducted to evaluate the control system at the system unit development, system integration development and operational system development stages shown in Figure 2: Methodology Stages.

The system unit development and evaluation stages will test the components of the control system in isolation. The unit tests will employ non-probability purpose sampling to collect system log file information during run-time. Each unit test will have an expected log file outcome, which is the criteria to determine the pass or fail result. Document analysis of the log files will then determine the outcome of the test according to the criteria. The unit tests of the system components will verify the successful operation of local device peripheral initialisation, master controller configuration, slave controller configuration, master control logic engine, slave control logic engine, JSON file generation, the separation of hardware and software through sub-routine APIs and data structure generation.

The integration development and evaluation stages will test the interactions between each control system relationship. Non-probability purpose sampling will record system component interactions in log files. Qualitative document analysis will then observe the resulting outcome from the run-time operations. The controlled test's expected outcome will determine each integration test's criteria. The result within the log file document will be assessed against the criteria to determine either pass or fail outcome. The integration tests will evaluate the communication configuration and initialisation, bi-directional communication between master and slave controllers, slave controller prioritisation and coordination, slave controller peripheral MCU pin states and master-slave command data structure parsing.

The system development and evaluation stages will test the control system within its operational environment. Start-to-finish system testing will be conducted during this stage by implementing an operational escape room. Multiple slave controllers will be configured as escape room puzzles, each with its control logic and

configuration files. Each puzzle will test different types of input and output configurations to validate the ability of the escape room to interface with different data configurations. A single master controller will have its configuration file defined with control logic, which requires hybrid progression logic paths. Purpose sampling will be employed within this stage to analyse the control system and its operational conditions. Log files covering sub-routine calls, test metrics, communication dump, control logic evaluation, and error handling will be generated during each system test. Document analysis will determine system performance and success according to the expected outcomes of the generated control logic and system configuration.

3.1.2. List of Materials

The materials and resources required to complete the build development stages and test regime are outlined within Appendix B: Budget and List of Materials.

3.1.3. Risk Assessment

Appendix A: Risk Assessment outlines the risks identified and mitigated for the build development stages and test regime.

3.1. Development Process

The stages of the development process are outlined in numerical order in the sections below. The development process will first design and verify the hardware and firmware components of the control system in isolation. Once the system components have passed isolated unit testing, the control system hardware and firmware, which connects integrated components, will be developed. The integration of control system components will be evaluated by conducting integration testing for each system relationship. Once the system is integrated, it will be tested in its operational environment. Multiple slave controller puzzles will be created, each with a defined configuration schema on their respective SD card. The controller will have its configuration schema and control logic data structure uploaded to its SD card. A range of system testing will be conducted to evaluate the operational objectives and outcomes of the experiment. Upon successful system testing, the control system will undertake stress testing to evaluate the system's performance characteristics and limitations.

3.1.1. Embedded Architecture of Master Controller and Slave Controller.

Design Rationale

The master controller circuitry will require the hardware components shown in Figure 2: Master Controller Architecture. The master controller hardware will consist of a Bluetooth BLE interface, clocking circuitry, SD card interface, power supply electronics, microcontroller programming interface and LED status feedback indicators. The ESP32-C6-DevKit-1-N8 is a development kit for the ESP32-C6 and contains the necessary hardware for Bluetooth BLE antenna, 5V and 3.3V voltage regulation, clocking circuitry and an LED indicator. Therefore, the devkit was selected as the embedded hardware selection for both the master and slave controller. The ESP-IDF development environment has library components for SD card interfacing using SPI. Therefore, the ESP32-C6-DevKit-1-N8 will interface with the SD card storage through the sdmmc component within ESP-IDF.

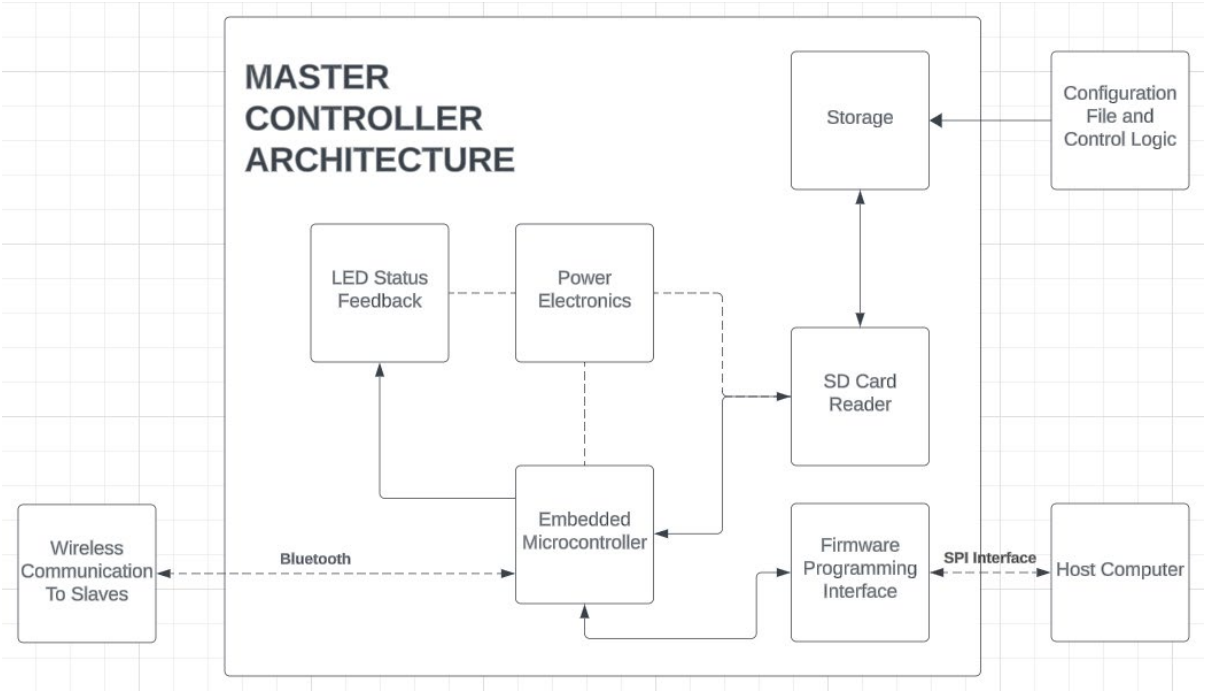


Figure 2: Master Controller Architecture

The slave controller circuitry will require the hardware components shown in Figure 3: Slave Device Architecture. The slave controller hardware will have a Bluetooth BLE interface, clocking circuitry, SD card interface, power supply electronics, microcontroller programming interface, LED status feedback and peripheral MCU input/output interface.

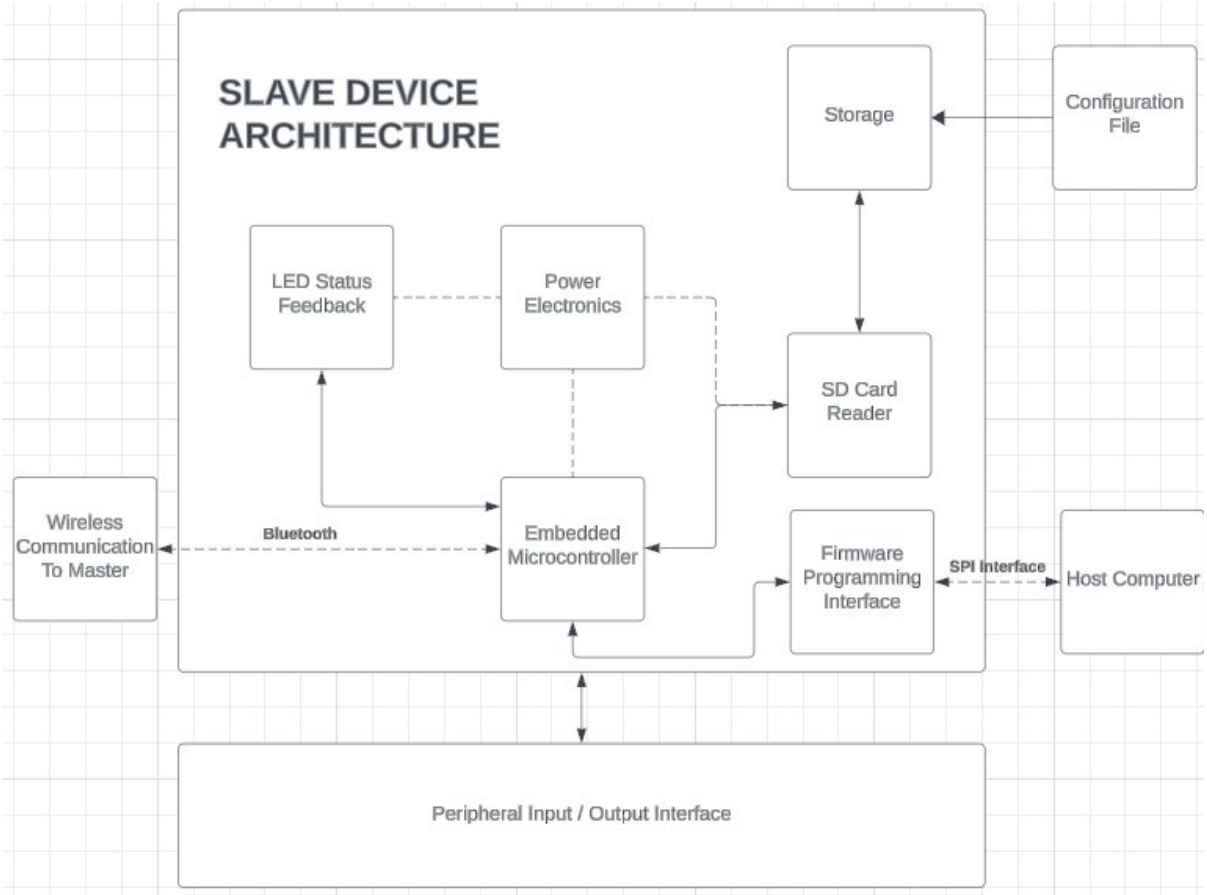


Figure 3: Slave Device Architecture

Component Requirements

The firmware of the master controller embedded system will require multiple interfacing routines to enable controller peripheral functions. The first routine, sd_card_interface, will interface with the embedded microcontroller and SD card storage. The sd_card_interface routine will enable read, write, append, mounting and unmounting operations. The second routine, led_hmi_interface, will involve toggling the state of the human-machine LED indicator. The functionality of these sub-routines and the master embedded controller will be evaluated through Table 1: Unit Test UT01.

The firmware of the slave controller will make use of the same sd_card_interface and led_hmi_interface routines as the master controller. The peripheral_update routine will query the peripheral microcontroller pins to set output pin state values and read discrete and analogue input pin values. The functionality of these sub-routines and the embedded slave controller will be evaluated through Table 2: Unit Test UT02.

Relevant Objectives and Outcomes

- Design of low-cost master-slave control system architecture suitable for escape room puzzles.

3.1.2. Embedded JSON Serialiser

Design Rationale

The communication and configuration of the embedded control system will require a file format representing or containing the data of run time internal data structures. The review of the literature in section 2.4.2 has resulted in the JSON file format being selected as the communication and configuration file type standard. JSON was selected as it can be used for communication payloads and configuration files, enabling a single file format to standardise both. The JSON file format is also less verbose and easier to read than the XML format. This assists in developing a universal game progression logic file for managing escape room progression.

Component Requirements

Log file documents and communication payloads must be constructed within the embedded system controllers for dynamic runtime monitoring and communication. This functionality will require the development of embedded JSON serialisation routines. The functional specification of the embedded JSON serialiser is to ingest a target JSON schema as a parameter and construct the JSON file according to the schema definition. The embedded JSON serialiser should support all data types and structures allowed within the JSON syntax definition. Once the JSON file has been constructed, the memory location will be returned to the function that called it for further processing.

Relevant Objectives and Outcomes

This specification will assist in achieving the following objectives and outcomes.

- Design of low-cost master-slave control system architecture suitable for escape room puzzles.
- Create a flexible game progression logic file type and data structure that can be loaded onto the master and slave controllers via SD card.

- Implementation of master and slave interpreter for the game progression logic file data structure. Ensuring accurate game progression and puzzle state management.
- A game progression logic file interpreter on both master and slave devices, ensuring correct puzzle state management, game progression and data transfer.

Implementation Details

Three components are necessary to achieve the specification for the JSON file serialiser. The first will be a json_file_serialiser routine, ingesting a target JSON schema file name. The json_schema_serialiser routine will traverse the JSON schema file and construct the resulting JSON file according to the schema definition. The second routine necessary will be get_schema_content, which will return the run-time data values for the generated JSON file's property value pairs. The get_schema_content routine will navigate the internal data structures and return specific values depending on the schema, property and object index key parameters. The target JSON file schemas will be the final component for the embedded JSON seraliser. A JSON schema file must be defined and stored on the associated controller's SD card. A distinct JSON file schema will need to be provided for each JSON file the system can generate. The functionality of each component for the embedded JSON serialiser will be evaluated according to Table 3: Unit Test UT03.

3.1.3. Bluetooth BLE Connection Interface

Design Rationale

The selection of an appropriate wireless technology for the master-slave architecture was made according to the technical specifications in the literature review section Communication Strategies for Escape Room Control System and ease if integration. Bluetooth BLE is the selected technology as it's integrated into the selected ESP32-C6 chipset from the embedded system design.

Component Requirements

Both the master controller and slave controller devices will require a Bluetooth BLE connection interface sub-routine. The connection interface functions will handle data transfer between master and slave controllers, establish the connection between the master controller and target slave controller, disconnect the target slave controller from

the master controller and manage multiple slave controller connection instances within the master controller. The functionality of these sub-routines and functions will be evaluated through Table 4: Unit Test UT04.

Relevant Objectives and Outcomes

- Evaluate and implement appropriate wireless communication methods, protocols and technologies for the master and slave controllers.
- Design and develop Bluetooth connection interface for the master controller to manage multiple slave controller connections.
- Reliable data transfer between master controller and slave controller.

3.1.4. Master Controller Configuration

The configuration functionality of the master controller device will enable the separation of hardware and control system software. The configuration routines of the master controller validate the mounted JSON files against three JSON schema file definitions. The configuration file schema will define the system log file structure, device profile schema, and communication profile definitions. The configuration file schema is required to support scalable representations of communication, device, and session profiles. The master control logic schema will define how control logic is represented in JSON format. The master control logic schema will define logic expression syntax, progression stage object structure, valid parameter data types, and valid logic expression operations. The structure of the master control logic schema will also allow for the representation of hybrid logic progression within the progression logic stages and their internal logic expressions. This will enable the complex logic expression paths within each progression logic stage. The final configuration schema necessary is the command message schema. The command message schema defines the command message structure for specific devices. This ensures the target controller can parse and interpret the communicated data packet into executable commands. The command message schemas define valid, executable commands and payload values for the target controller device.

The master controller's JSON configuration parser will read the configuration files mounted through the master controller's SD card interface sub-routines. The parser then deconstructs and transforms the JSON file into internal data structures. The

master controller configuration parser will detect and handle invalid JSON syntax or structure errors. The functionality of the master JSON configuration parser will be evaluated against Table 5: Unit Test UT05.

The final component responsible for the master controller configuration is the master controller's configuration file interpreter. This routine interprets the parsed configuration internal data structures and sets up operational communication settings, device profiles, control logic progression stages and session parameters. The interpreter initialises and validates the system context of the master controller. The master controller's configuration file interpreter logs the outcome of each stage during the initialisation of the master controller. The interpreter indicates the outcome of the master controller configuration and initialisation to the LED HMI. The routine handles errors during configuration and indicates the HMI's error type. The functionality of the master controller's configuration file interpreter will be evaluated against Table 6: Unit Test UT07.

3.1.5. Slave Controller Configuration

The configuration functionality of the slave controllers also enables the separation of hardware and control system software. Like the master controller, the configuration of the slave controllers validates the mounted JSON files against three JSON schema file definitions. The slave controller's configuration file schema will define the system log file structure, session profiles, device definition profile, peripheral microcontroller definition and communication device profiles. The peripheral microcontroller definition will contain the valid pinout compliance of the peripheral microcontroller. This defines the valid pin directions, signal types and pin mapping for each pin. The slave control logic schema defines how control logic is represented in JSON format. The scope of the control logic expressions and parameters are local to the slave controller's peripheral MCU pin state. The slave control logic schema will define valid logic expression syntax, progression stage object structure, valid parameter data types, and valid logic expressions. The structure of the slave control logic schema will also allow for the representation of hybrid logic progression within the progression logic stages and their internal logic expressions. This will enable the complex logic expression paths within each progression logic stage. The final configuration schema necessary is the command message schema. The command message schema defines the

command message structure for specific devices. This ensures the target controller can parse and interpret the communicated data packet into executable commands. The command message schemas define valid, executable commands and payload values for the target controller device.

The slave controller JSON configuration parser will read the configuration files mounted through the SD card interface sub-routine. The parser will then deconstruct and transform the JSON file into internal data structures. The slave controller configuration parser will detect and handle invalid JSON syntax or structure errors. The functionality of the slave JSON configuration parser will be evaluated against Table 7: Unit Test UT08.

The slave controller's configuration file interpreter executes the configuration settings according to the parsed configuration files. The interpreter initialises communication settings, device profiles, control logic progression stages and session parameters related to the slave controller. The configuration file interpreter can handle errors while initialising the slave controller's configuration and record such functions' outcomes to log files. The outcome of the slave controller's configuration is indicated on the device's HMI and recorded on SD card log flies. The functionality of the slave controller's configuration file interpreter is evaluated against Table 8: Unit Test UT10.

3.1.6. Master Controller's Control Logic Engine

The master controller's control logic engine encapsulates the components responsible for managing system context for the current progression logic stage, managing progression logic state transitions, parsing control logic file, interpreting control logic internal data structures, evaluating control logic expressions and dispatching actions according to control logic evaluation outcome. The scope of the master controller's control logic engine is to manage system-wide control logic evaluation across one-to-many slave controllers. This requires that the control logic expressions can represent parameters across multiple slave control devices.

The master controller's control logic parser deconstructs and transforms the JSON control logic file into internal data structures. The internal data structures must represent scalable hybrid control logic expressions and the system context for each

progression logic stage. The master controller's JSON control logic parser's functionality will be evaluated against Table 11: Unit Test UT11.

The master controller's control logic interpreter evaluates the outcome of logic expressions that ingest parameters across multiple slave controller devices. The master control logic interpreter will ingest the parameters for the current progression logic stage according to the system context definition. These parameters will then be utilised to evaluate the current progression logic stage outcome. The outcome of the control logic expression will then be returned to the master controller's control logic action dispatcher to operate according to the result. Each control logic evaluation's system context and outcome are recorded on SD card storage log files. The functionality of the master controller's control logic interpreter will be evaluated against Table 10: Unit Test UT12.

The master controller's state manager updates the internal data structures and system context according to transitions in slave controller parameters and the progression logic stage. The state manager ingests the state values relevant to the current progression logic stage expressions, manages the transition of progression logic stages and updates the system context according to updates. The functionality of the master controller's state manager will be evaluated against Table 11: Unit Test UT13.

The final component within the master controller's control logic engine is the control logic action dispatcher. The action dispatcher coordinates the control logic engine's sub-routines according to system context and logic expression evaluation outcome. The master controller's action dispatcher initiates the transaction of command messages to target slave controllers. The action dispatcher coordinates the parsing and interpretation of the current progression logic stage. It is also responsible for coordinating the progression logic stage transitions by calling the SD card interface routines and the master controller's state manager. The functionality of the master controller's action dispatcher will be evaluated against Table 12: Unit Test UT14.

3.1.7. Slave Controller's Control Logic Engine

The slave controller's control logic engine encapsulates the components responsible for managing system context for the current progression logic stage, managing

progression logic state transitions, parsing control logic file, interpreting control logic internal data structures, evaluating control logic expressions and dispatching actions according to control logic evaluation outcome. The scope of the slave controller's control logic engine is to manage control logic local to the slave controller's peripherals. This requires that the logic expressions trigger when the local slave controller's peripherals evaluate to the current progression logic stage expression for the specific slave device.

The slave controller's control logic parser deconstructs and transforms the JSON control logic file into internal data structures. The internal data structures need to represent logic expressions and trigger values for the peripherals of the slave controller. The control logic internal data structures also represent the system context for the peripheral MCU pin configuration for the current progression logic stage. The parser will be able to handle invalid JSON file syntax and structure. The slave controller's JSON control logic parser's functionality will be evaluated against Table 13: Unit Test UT15.

The slave controller's control logic interpreter evaluates the outcome of logic expressions local to the device's peripherals. This requires that the interpreter ingests the state of multiple peripherals set by the system context for the current progression logic stage. These parameters will be evaluated against the trigger conditions within the current progression logic stage's logic expressions. The outcome of the control logic expressions will then be returned to the slave controller's control logic action dispatcher to operate according to the result. Each control logic evaluation's system context and outcome are recorded on SD card storage log files. The functionality of the master controller's control logic interpreter will be evaluated against Table 14: Unit Test UT16.

The master controller's state manager is responsible for updating the internal data structures and system context according to transitions in state of the peripheral MCU and progression logic stage. The state manger ingests the current logic expression parameter state values, manages the transition of progression logic stages and updates the system context according to the updates. The functionality of the slave controller's state manager will be evaluated against Table 15: Unit Test UT17.

The local control logic action dispatcher coordinates the slave controller's control logic engine. The action dispatcher coordinates the control logic engine's sub-routines according to system context and logic expression evaluation outcome. The slave controller's action dispatcher also initiates communication back to the master controller once the current progression logic stage expressions are evaluated to be true. The action dispatcher is responsible for coordinating the parsing, interpretation of internal data structure, logic evaluating and progression logic stage transitions. The functionality of the slave controller's action dispatcher will be evaluated against Table 16: Unit Test UT18.

3.1.8. Command Message Communication

The command message communication sub-routines are responsible for processing outbound and inbound communication between the master controller and the target slave controller. The device receiving the communication transmission will parse, validate, interpret and dispatch actions according to the state of the command message received. The controller transmitting the command message will construct and serialise the JSON command message according to internal data structures and target command message JSON schema.

The master controller's communication action dispatcher will be responsible for coordinating and prioritising slave controller message requests, calling command message sub-routines to process the requests, recording all communication transactions to log file SD card storage and coordinating the transmission of command messages to the target slave controller. The functionality of the master controller's communication action dispatcher will be evaluated against Table 17: Unit Test UT19.

The slave controller's communication action dispatcher will be responsible for coordinating received command messages from the master controller, calling command message sub-routines to process the requests, recording all communication transactions to log file SD card storage and coordinating the transmission of command messages to the master controller. The functionality of the slave controller's communication action dispatcher will be evaluated against Table 18: Unit Test UT20.

The command message parser will ingest a target JSON command message and transform it into internal data structures. The command message parser will detect and handle invalid JSON syntax and structure errors. The functionality of the command message parser will be evaluated against Table 19: Unit Test UT21.

The master controller's command message interpreter evaluates the command and payload of the received command message. Depending on the received command message, the master controller's communication interpreter will call associated action dispatcher sub-routines to process the message payload. The interpreter will ingest the parsed and validated command message data structures to determine the required action. The functionality of the master controller's command message interpreter will be evaluated against Table 20: Unit Test UT23.

The slave controller's command message interpreter evaluates the command and payload of the received command message. Depending on the received command message, the slave controller's communication interpreter will call associated action dispatcher sub-routines to process the message payload. The interpreter will ingest the parsed and validated command message data structures to determine the required action. The functionality of the slave controller's command message interpreter will be evaluated against Table 21: Unit Test UT24.

3.1.9. Master Controller's and Slave Controller's Communication Engine

The integration between the master controller's communication action dispatcher and the slave controller's communication action dispatcher involves transferring and processing command messages between the master controller and multiple slave controller devices. The scope of the integration test is to validate the correct coordination of multiple slave controllers sending command messages. The master controller will sequentially transfer each type of the CommandMessage commands to a slave controller. Once the final CommandMessage is received from the master controller, the slave controller will then sequentially transfer each type of CommandMessage command to the master controller. Each controller will acknowledge data transmission before the transfer begins. The functionality between the master controller's communication action dispatcher and the slave controller's action dispatcher will be evaluated against Table 22: Integration Test IT03.

3.1.10. Master Controller Initialisation and Slave Controller Initialisation.

The integration between the master controller initialisation routine and the slave controller initialisation routines involves querying the initialisation status of the slave controller device profiles after the master controller configuration. The master controller validates the initialisation status of all device profiles configured within the system context. The master controller waits until all slave controllers have responded with an initialisation command message containing a successful configuration payload. This ensures that the entire control system has been configured and initialised. The functionality between the master controller's initialisation and slave controller initialisation is evaluated against Table 23: Integration Test IT04.

3.1.11. Master Controller Configuration and Control Logic Engine Integration.

Once the master controller has been initialised and the system context has been configured, the control logic engine is called. The control logic engine will parse and interpret the context of the first progression logic stage and indicate the status of the escape room game session to the HMI and log files. The functionality between master controller initialisation and control logic engine is evaluated against Table 24: Integration Test IT05.

3.1.12. Master Controller's Control Logic Action Dispatcher and Communication Action Dispatch Integration

Integrating the master controller's control logic action dispatcher and communication action dispatcher involves the two components calling each other sub-routines according to the command message and control logic outcome. The master controller's action dispatcher will call the communication action dispatcher to update the progression logic stage of slave controllers during stage transitions. The control logic action dispatcher will also request read and write state updates to target slave controllers depending on control logic expression evaluation. The communication action dispatcher will call the control logic action dispatcher when the current progression logic stage parameter has been sent an updated payload from relevant slave controllers. The functionality between the master controller's control logic action

dispatcher and communication action dispatcher integration is evaluated against Table 25: Integration Test IT06.

3.1.13. Slave Controller's Control Logic Action Dispatcher and Communication Action Dispatch Integration

The integration between the master controller's control logic action dispatcher and communication action dispatcher involves the two components calling each other's sub-routines according to the command message and control logic outcomes. The control logic action dispatcher will call the communication action dispatcher when the control logic expression for the current progression logic stage is evaluated as true. The control logic action dispatcher will call the communication action dispatcher when processed command message requests require a status response. The communication action dispatcher calls the control logic action dispatcher when updated progression logic stage command messages are received from the master controller. The communication action dispatcher may call the control logic action dispatcher when the master controller sends command messages requesting peripheral updates. The integration functionality between the slave controller's control logic action dispatcher and communication action dispatcher are evaluated against Table 26: Integration Test IT07.

3.1.14. Operational Escape Room Purpose Testing

With the control system passing both unit and integration testing, it can be developed to meet its operational conditions. The control system, configuration management, and data structures will undergo system testing to evaluate an escape room's functional objectives and outcomes. Three slave controller boards will be developed with differing peripheral input and output configurations. The first slave controller will sample analogue input values across three input pins of the peripheral microcontroller. The peripheral microcontroller for the first slave controller will also have four LEDs as puzzle output. The second slave controller will sample digital input values from a keypad. The puzzle will also involve a virtual buffer that manages the state of a string value buffer. The second slave controller will have two LEDs as puzzle output. Finally, the third puzzle will be developed to ingest the analogue values of two hall effect sensors. The third puzzle will have three LEDs as output to indicate the presence of a magnet. The master controller configuration, control logic, and command message

profile files will then be developed for a single escape room narrative. The progression logic stages within the control logic file will test both sequential and hybrid progression paths. Each progression logic stage will incorporate the developed slave controllers utilising different logic expressions at each stage. The slave controller configuration, control logic, and command message profile files will then be developed for the session narrative. The progression logic stages within the control logic file will evaluate varying logic expressions that modify both the peripheral MCU input and output pins. The configuration files will be mounted into the master and slave controllers, and the devices will be powered on. The escape room puzzles will then be interacted with, completing each progression logic stage as the system logs its response to SD card storage. The operational escape room purpose test will be evaluated against Table 27: System Test ST01.

3.1.15. Operational Escape Room User Purpose Testing

Once successful evaluation of operation escape room purpose testing from Table 27: System Test ST01, human users will test the escape room control system. The same three slave controllers from Operational Escape Room User Purpose Testing will be utilised within this system test. The master controller configuration, control logic, and command message profile files will then be developed for a new escape room narrative. The progression logic stages within the control logic file will test both sequential and hybrid progression paths. Each progression logic stage will incorporate the developed slave controllers utilising different logical expressions at each stage. The slave controller configuration, control logic, and command message profile files will then be developed for the second session narratives. The progression logic stages within the control logic file will evaluate varying logic expressions that modify both the peripheral MCU input and output pins. Three separate user tests will be conducted to demonstrate the versioning of session management profiles within the escape room control system. For each user test session, the principal user will be briefed on how to interact with the system safely, and narrative objectives will be explained. The configuration files will be mounted into the master controller and slave controllers, and the devices will be powered on. The principal user will then interact with the escape room puzzles, completing each progression logic stage as the system logs its response to SD card storage. The operational escape room purpose test will be evaluated against Table 28: System Test ST02.

3.1.16. Operational Communication Coverage Performance Testing

The quantitative experiment aims to determine the operational communication range between master controller and slave controllers under various operational conditions. The master controller will be placed in a fixed location inside a house. The communication coverage routine will then be developed and flashed onto the master and slave controllers. The master controller's communication coverage performance test sub-routine will establish a connection with the slave controller, send a single status update command message to the slave controller every 30 seconds for 10 minutes and then disconnect. The slave controller will need to receive, parse, validate, interpret and respond to all the master controller's command message transmissions. The master controller's communication coverage performance test sub-routine will then parse, validate and interpret the response from the slave controller to determine if the message was received correctly. The slave controller will move one metre away from master controller after each sub-routine test is completed. The communication coverage performance test will be repeated from one meter to fifty meters away from the master controller's fixed location. The message packets received from the master and slave controller will be recorded in the SD card storage log files. At each incremental increase in distance, the RSSI value will be recorded for both the master and slave controller. The performance test will be evaluated using Table 29: Performance Test PT01. From the recorded log files, the RSSI, connection success, and error rate can be analysed against the slave controller distance.

3.1.17. Operational Communication Latency Performance Testing

This performance test aims to determine the operational communication latency between the master controller and slave controller command message requests. The operational latency test will determine the time taken for the slave controller to acknowledge the master controller's message commands with a command message response. The communication latency sub-routine will be flashed onto the master controller, which will interface with the ESP32-C6 development board's internal timer interrupts. The master controller will initialise the internal timer to zero before sending each command message to the slave controller. Once the master controller receives the response from the slave controller, the internal timer interrupt is triggered, which records the value of the timer the moment the command message response is

received. The command messages sent from the master controller will establish the connection, query peripheral status updates and disconnect. The performance test will be evaluated using Table 30: Performance Test PT02.

3.1.18. Concurrent Slave Controller Communication to Master Controller Performance Testing

This operational performance test aims to evaluate the coordination and scheduling limitations of concurrent slave controller communication. Three slave controller boards will be developed, with the slave controller sub-routines flashed onto them. Each slave controller board will have a unique command message payload response to send to the master controller. The master controller will be loaded with a control logic progression file with three progression logic stages. Each progression logic stage incrementally includes an additional slave controller until all three slave controllers are set for the final system context. The slave controller's logic state will configured to immediately respond to the master controller when it's included in the system context of the current progression logic stage. The slave controller will continue to retry its command message response until the master controller acknowledges its response. An additional concurrent slave controller response test sub-routine will be flashed onto the master controller, which will configure and set the internal timers of the ESP32-C6 development board. The log files on the slave controller will record the number of command message response retries, errors and processing time for each progression logic stage until all progression logic stages are completed. The master controller will record the number of concurrent connections, processing times, communication command message dump, sub-routine calls and communication events for each progression logic stage. The performance test will be evaluated using Table 37: Performance Test PT03.

3.1.19. Concurrent Control Logic Evaluation Performance Testing

The performance test is designed to evaluate the performance of concurrent control logic parameters within the control logic expressions. The performance evaluation will record the processing time and utilisation factor of the system resources within both the master controller and slave controller devices as scale of control logic expressions increase. The performance test will require three slave controllers to be flashed with the slave controller routines. The master controller and slave controllers will have

configuration and control logic files developed, incrementally increasing the parameter complexity of the control logic expressions in each progression logic stage. Each progression logic stage will require an additional peripheral to be evaluated within each slave controller device. The control logic expressions within the master controller will incrementally require additional slave controller device peripherals to be evaluated within its logic expressions. At each progression logic stage, internal timer interrupts will record the time taken during each sub-routine process of the master and slave controller devices. Memory and storage utilisation will also be recorded at each progression logic stage. The performance test will be evaluated using Table 35: Performance Test PT04.

3.2. Testing Regime

3.2.1. Unit Testing

The tables within the unit testing section outline the unit testing, which will be conducted to evaluate the performance of each component to achieve the outcomes and objectives. Each unit test verifies the functionality of individual components in isolation.

Table 1: Unit Test UT01

Test Case ID	UT01
Relevant Test Objective	• Master controller architecture and interfacing.
Test Steps	1. Design master controller embedded system. 2. Assemble master controller embedded system according to schematic. 3. Flash embedded system interfacing sub-routines onto master controller. 4. Define sub-routine interfacing sequence and output as expected outcome for master controller embedded system. 5. Format SD card with FAT32 file system and insert into master controller SD card slot. 6. Power on master controller embedded system and generate log file documents.

	7. ESP32-C6 development board LED flashing indicates generated log files stored to SD card and unmounted from file system. 8. Power off development board and eject SD card. 9. Store generated log files onto local host computer for document analysis.
Data Collected	1. unit01.txt: Text log file containing ouput of sd_card_interface function operations.
Pass/Fail Criteria	Pass • The content within unit01.txt file matches write and append expressions of main.c for unit test. • The content within unit01_read.txt file is exists within unit01.txt file.

Table 2: Unit Test UT02

Test Case ID	UT02
Relevant Test Objective	• Slave controller architecture and interfacing.
Test Steps	1. Design slave controller embedded system. 2. Assemble slave controller embedded system according to schematic. 3. Flash embedded system interfacing sub-routines onto slave controller microcontroller. 4. Define sub-routine interfacing sequence and output as expected outcome for slave controller embedded system. 5. Format SD card with FAT32 file system and insert into slave controller SD card slot. 6. Power on slave controller embedded system and generate log file documents. 7. ESP32-C6 development board LED flashing indicates generated log files stored to SD card and unmounted from file system.

	8. Power off slave controller and eject SD card. 9. Store generated log files onto local host computer for document analysis.
Data Collected	• unit01.txt: Text log file containing ouput of sd_card_interface function operations.
Pass/Fail Criteria	Pass • The content within unit01.txt file matches write and append expressions of main.c for unit test. • The content within unit01_read.txt file is exists within unit01.txt file.

Table 3: Unit Test UT03

Test Case ID	UT03
Relevant Component	• Embedded JSON serialiser
Test Procedure	1. Develop json_file_serialiser and get_schema_content routines in the programming language C. 2. Develop a JSON schema that includes various JSON data types. 3. Connect and develop interface routines for ESP32-C6 development kit hardware SD card reader. 4. Flash ESP-IDF MasterController project contains json_file_serialiser and get_schema_content routines the ESP32-C6 using host computer. 5. Format SD card with FAT32 file system. 6. Load JSON schema onto SD card storage from host computer. 7. Insert into master controller SD card slot and power on ESP32-C6 development board.

	8. The main.c routine will call json_file_seriliaser with the schema file name, and then the resulting JSON file will be written to sd card storage. 9. ESP32-C6 development board LED flashing indicates that the generated JSON file is stored and the SD card is unmounted from the file system. 10. Power off the development board and eject the SD card. 11. Store generated JSON file onto local host computer for document analysis.
Data Collected	1. json_file_serliaiser and get_schema_content source code. 2. Target json file schema. 3. Generated json file. 4. Generated JSON File. 5. Expected JSON file output. 6. Loaded JSON schema.
Pass/Fail Criteria	• The generated JSON file matches the structure and content within the associated JSON schema. • The generated JSON files are stored in the mounted SD card.

Table 4: Unit Test UT04

Test Case ID	UT04
Relevant Test Objective	• Bluetooth BLE connection interface.
Test Steps	1. Develop bluetooth_connection_interface sub-routine for both master controller and slave controller. 2. Flash bluetooth_connection_interface sub-routine onto master controller embedded ESP32. 3. Flash bluetooth_connection_interface sub-routine onto slave controller embedded ESP32.

	4. Define expected outcome from generated log files. 5. Format SD card with FAT32 file system and insert into slave controller SD card slot. 6. Power on master controller and two slave controller boards. 7. Master controller establishes connection between two slave controller devices. 8. Master controller disconnects from one of the slave controllers. 9. Master controller reconnects to slave controller. 10. Master controller sequentially transfers payload to both connected slave controller devices. 11. Slave controller received data transmission triggers interrupt service routine to handle data reception. 12. Master controller and both slave controller boards HMI LED flashing indicates generated JSON file stored, and SD card unmounted from file system. 13. Power off master controller and both slave controller boards and eject SD cards. 14. Store generated JSON log files onto local host computer for document analysis.
Data Collected	1. Sub-routine call log file: Records the sequence of sub-routines calls. 2. Control system error handling log file: Caught run-time errors. 3. Methodology test data log file: Outcome established connections, outcome of transferred payloads and disconnection. 7. Communication dump log file: Transmitted and received payloads. Transmitting device GUID, Receiving device GUID.
Pass/Fail Criteria	• The sub-routine call log file matches expected outcome defined during implementation.

	• The control system error handling log file matches expected outcome defined during implementation. • The methodology call log file matches expected outcome defined during implementation. • The communication dump log file matches expected outcome defined during implementation.

Table 5: Unit Test UT05

Test Case ID	UT05
Relevant Test Objective	• Master controller JSON configuration parser. • Ensure separation of hardware and software implementation through JSON configuration files. • Scalable representation of communication profiles, session profiles and device profiles.
Test Steps	1. Develop and flash master json configuration parser routine onto the master controller. 2. Define JSON configuration file schema. 3. Define JSON control logic schema. 4. Develop master controller configuration JSON file which is structured from defined schema with correct syntax. 5. Format SD card with FAT32 file system. 6. Load master controller configuration schema, master control logic schema and master communication profile schema onto SD card storage. 7. Load master controller configuration, control logic and command message JSON files onto SD card storage. 8. Define expected outcome from generated log files. 9. Format SD card with FAT32 file system and insert into master controller SD card slot. 10. Power on master controller board.

	11. Master controller parses JSON configuration files into internal data structure. 12. HMI interface flashing indicates all configuration files parsed, log files stored to SD card and SD card unmounted from file system. 13. Power off master controller and eject SD card storage. 14. Store generated JSON file onto local personal computer for document analysis.
Data Collected	1. Sub-routine call log file: Records the sequence of sub-routines calls. 2. Control system error handling log file: Caught run-time errors. 3. Methodology test data log file: Internal data structures generated from parsed JSON file. 4. Session management log file: Record the internal data structures generated for the current session.
Pass/Fail Criteria	Pass • Sub-routine call log file, control system error handling log file, methodology call log file, session management log file matches expected outcome defined during implementation.

Table 6: Unit Test UT07

Test Case ID	UT07
Relevant Test Objective	• Master controller JSON configuration interpreter.
Test Steps	1. Implement and flash master configuration file interpreter routine onto the master controller. 2. Staticly define system context data structures to indicate different configuration stages of slave controller device profiles.

	3. Develop configuration files with both valid and invalid structures according to target schemas. 4. Load master controller configuration file, master control logic configuration file and master communication profile file schemas to SD card. 5. Define the expected outcome from generated log files. 6. Format the SD card with the FAT32 file system and insert it into the master controller SD card slot. 7. Power on master controller board. 8. HMI interface flashing indicates all configuration files are parsed, validation is complete, log files are stored to the SD card, and the SD card is unmounted from the file system. 9. Power off the master controller and eject SD card storage. 10. Store generated JSON file onto a local personal computer for document analysis.
Data Collected	1. Sub-routine call log file: Records the sequence of sub-routines calls. 2. Control system error handling log file: Caught run-time errors. 3. Methodology test data log file: System context internal data structures. 4. Session management log file: Session profile internal data structure generated. Session system context generated. 5. Control system evaluation log file: System context parameters at time of evaluation.
Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file,

	control system evaluation log file matches expected outcome defined during implementation.

Table 7: Unit Test UT08

Test Case ID	UT08
Relevant Test Objective	• Slave controller JSON configuration parser. • Ensure separation of hardware and software implementation through JSON configuration files. • Scalable representation of communication profiles, session profiles and device profiles.
Test Steps	1. Develop and flash master json configuration parser routine onto the slave controller. 2. Define JSON configuration file schema. 3. Define JSON control logic schema. 4. Define JSON communication profile schema. 5. Develop a slave controller configuration JSON file that is structured from a defined schema with correct syntax. 6. Develop slave controller control logic JSON file which is structured from defined schema with correct syntax. 7. Format SD card with FAT32 file system. 8. Load slave controller configuration schema, master control logic schema and communication profile schema onto SD card storage. 9. Load slave controller configuration, control logic and command message JSON files onto SD card storage. 10. Define expected outcome from generated log files. 11. Format SD card with FAT32 file system and insert into slave controller SD card slot. 12. Power on master controller board.

	13. Slave controller parses JSON configuration files into the internal data structure. 14. HMI interface flashing indicates all configuration files parsed, log files stored to SD card and SD card unmounted from file system. 15. Power off slave controller and eject SD card storage. 16. Store generated JSON file onto local personal computer for document analysis.
Data Collected	• Sub-routine call log file: Records the sequence of sub-routines calls. • Control system error handling log file: Caught run-time errors. • Methodology test data log file: Internal data structures generated from parsed JSON file. • Session management log file: Record the internal data structures generated for the current session.
Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file, control system evaluation log file matches expected outcome defined during implementation.

Table 8: Unit Test UT10

Test Case ID	UT10
Relevant Test Objective	• Slave controller JSON configuration interpreter.
Test Steps	1. Implement and flash slave configuration file interpreter routine onto the slave controller. 2. Staticly define system context data structures to indicate different configuration stages of peripheral profiles. 3. Define the JSON configuration file schema for SD card storage.

	4. Develop configuration files with both valid and invalid structures according to target schemas. 5. Load slave controller configuration file, slave control logic configuration file and slave communication profile file schemas to SD card. 6. Define the expected outcome from generated log files. 7. Format the SD card with the FAT32 file system and insert it into the slave controller SD card slot. 8. Power on master controller board. 9. HMI interface flashing indicates all configuration files are parsed, validation is complete, log files are stored to the SD card, and the SD card is unmounted from the file system. 10. Power off the slave controller and eject SD card storage. 11. Store generated JSON file onto a local personal computer for document analysis.
Data Collected	• Sub-routine call log file: Records the sequence of sub-routines calls. • Control system error handling log file: Caught run-time errors. • Methodology test data log file: System context internal data structures. • Session management log file: Session profile internal data structure generated. Session system context generated. • Control system evaluation log file: Outcome of logic expression evaluation. System context parameters at time of evaluation.
Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file,

	control system evaluation log file matches expected outcome defined during implementation.

Table 9: Unit Test UT11

Test Case ID	UT11
Relevant Test Objective	• Master controller JSON control logic parser.
Test Steps	1. Develop valid control logic JSON files that conform to schema. 2. The first control logic file contains sequential logic progression stages with single logic operations within each stage. 3. The second control logic file contains sequential logic progression stages with logic operations with varying complexity in each stage. 4. The third control logic file contains hybrid logic progression stages with single logic operations within each stage. 5. The fourth control logic file contains sequential logic progression stages with logic operations with varying complexity in each stage. 6. Each control logic file will contain logical AND, OR and NOT logic operators. 7. Develop invalid control logic JSON files that do not conform to the schema. 8. The first control logic file contains sequential logic progression stages with single logic operations within each stage. 9. The second control logic file contains sequential logic progression stages with logic operations with varying complexity in each stage. 10. The third control logic file contains hybrid logic progression stages with single logic operations within each stage.

	11. The fourth control logic file contains sequential logic progression stages with logic operations with varying complexity in each stage. 12. Each control logic file will contain logical AND, OR and NOT logic operators. 13. Implement and flash the master controller JSON control logic parser sub-routine onto the master controller. 14. Load developed JSON control logic files onto the SD card. 15. Load developed configuration file schemas onto the SD card. 16. Mount the SD card into the master controller board and turn power on to the controller circuit. 17. HMI interface flashing indicates all control logic files are parsed, log files are stored on an SD card, and the SD card is unmounted from the file system. 18. Power off the master controller and eject the SD card from the controller. 19. Store generated JSON log files onto a local personal computer for analysis.
Data Collected	• Sub-routine call log file: Records the sequence of sub-routines calls. • Control system error handling log file: Caught run-time errors. • Methodology test data log file: The parsed internal data structure as property-value pairs. • Control system evaluation log file: Outcome of logic expression evaluation. System context parameters at time of evaluation. • Unit test log files record internal data structures which store control logic parameters for each progression logic stage.

	• Unit test log files record logic expression constructed for each progression logic stage. • Unit test log files record system context at each progression logic stage. • Unit test log files record errors caused by invalid control logic files. • Unit test log files record run time errors.
Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file, control system evaluation log file matches expected outcome defined during implementation.

Table 10: Unit Test UT12

Test Case ID	UT12
Relevant Test Objective	• Master controller JSON control logic interpreter.
Test Steps	1. Develop valid control logic JSON files that conform to the schema. 2. The first control logic file contains sequential logic progression stages with single logic operations within each stage. 3. The second control logic file contains sequential logic progression stages with logic operations with varying complexity in each stage. 4. The third control logic file contains hybrid logic progression stages with single logic operations within each stage. 5. The fourth control logic file contains sequential logic progression stages with logic operations with varying complexity in each stage. 6. Each control logic file will contain logical AND, OR and NOT logic operators.

	7. Implement and flash master control logic interpreter sub-routines onto the master controller. 8. Load developed JSON control logic files onto the SD card. 9. Load developed configuration file schemas onto SD card. 10. Insert the SD card to the master controller board and turn the power on to the controller circuit. 11. HMI interface flashing indicates all control logic files are parsed, logic expression evaluated, log files are stored on an SD card, and the SD card is unmounted from the file system. 12. Power off the master controller and eject SD card from the controller. 13. Store generated JSON log files onto a local personal computer for analysis. 14. HMI interface flashing indicates all control logic files are parsed, log files are stored on an SD card, and the SD card is unmounted from the file system. 15. Store generated JSON files onto a local personal computer for analysis.
Data Collected	• Sub-routine call log file: Records the sequence of sub-routines calls. • Control system error handling log file: Caught run-time errors. • Methodology test data log file: Internal data structure representation of logic expressions before and during evaluation. • Control system evaluation log file: Outcome of logic expression evaluation. System context parameters at time of evaluation. • Unit test log files record logic expression constructed for each progression logic stage.

	• Unit test log files record system context at each progression logic stage. • Unit test log files record system context parameter values during interpreter evaluation. • Unit test log files record internal data structures that store control logic expressions and parameters for each progression logic stage. • Unit test log files record internal data structures that store control logic parameters for each progression logic stage. • Unit test log files record the outcome of logic expression evaluation at each progression logic stage. • Unit test log files record errors caused by invalid control logic files. • Unit test log files record run time errors.
Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file, control system evaluation log file matches expected outcome defined during implementation.

Table 11: Unit Test UT13

Test Case ID	UT13
Relevant Test Objective	• Master controller JSON control logic state manager.
Test Steps	1. Develop valid control logic JSON files that conform to schema. 2. The first control logic file contains sequential logic progression stages with single logic operations within each stage.

	3. The second control logic file contains sequential logic progression stages with logic operations with varying complexity in each stage. 4. The third control logic file contains hybrid logic progression stages with single logic operations within each stage. 5. The fourth control logic file contains sequential logic progression stages with logic operations with varying complexity in each stage. 6. Each control logic file will contain logical AND, OR and NOT logic operators. 7. Each control logic file progression stage requires different system context configuration. 8. Develop and flash master controller's logic state manager sub-routines onto the master controller. 9. The main routine evaluates each expression as true, causing the state manager to transition to the next progression stage. System context variables are statically declared for updates after stage transitions. 10. Load developed JSON control logic files onto the SD card. 11. Load developed configuration file schemas onto the SD card. 12. Mount the SD card into the master controller board and turn power on to the controller circuit. 13. HMI interface flashing indicates all control logic files are parsed, the progression logic stage transitioned, log files are stored on an SD card, and the SD card is unmounted from the file system. 14. Power off the master controller and eject the SD card from the controller. 15. Store generated JSON log files onto a local personal computer for analysis.

Data Collected	• Sub-routine call log file: Records the sequence of sub-routines calls. • Control system error handling log file: Caught run-time errors. • Methodology test data log file: • Session management log file: • Control system evaluation log file: Outcome of logic expression evaluation. System context parameters at time of evaluation. • Unit test log files record system context of master controller at each progression logic stage. • Unit test log files record system context parameter values of relevant slave controllers for the current control logic progression stage. • Unit test log files record internal data structures which store control logic expressions and parameters for each progression logic stage. • Unit test log files record outcome of logic expression evaluation at each progression logic stage. • Unit test log files record system state and context transitions. • Unit test log files record errors caused by invalid control logic files. • Unit test log files record run time errors.
Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file, control system evaluation log file matches expected outcome defined during implementation.

Table 12: Unit Test UT14

Test Case ID	UT14

Relevant Test Objective	• Master controller JSON control logic action dispatcher.
Test Steps	1. Load onto SD card the configuration file schema necessary for configuration. 2. Develop valid control logic JSON files which conform to master control logic schema. Some which involve sequential logic progression. Some which involve hybrid logic expression paths. Distinct progression logic stages with different system context profiles. 3. Implement and flash master control logic action dispatcher sub-routines onto the master controller. 4. Load developed JSON control logic files onto SD card. Mount the SD card into master controller board and turn power on to the controller. 5. HMI interface flashing indicates all control logic files parsed, log files stored to SD card and SD card unmounted from file system. 6. Store generated JSON file onto local personal computer for analysis.
Data Collected	• Sub-routine call log file: Records the sequence of sub-routines calls. • Control system error handling log file: Caught run-time errors. • Control system evaluation log file: Outcome of logic expression evaluation. System context parameters at time of evaluation. • Unit test log files record sub-routine calls according to control system state and triggers. • Unit test log files record action dispatcher function calls. • Unit test log files records JSON files read from SD card.

	• Unit test log files record system state and context transitions. • Unit test log files record internal data structures generated from JSON file parsing. • Unit test log files record system context at each progression logic stage. • Unit test log files record system context parameter values during interpreter evaluation. • Unit test log files record internal data structures which store control logic expressions and parameters for each progression logic stage. • Unit test log files record outcome of logic expression evaluation at each progression logic stage. • Unit test log files record errors caused by invalid control logic files.
Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file, control system evaluation log file matches expected outcome defined during implementation.

Table 13: Unit Test UT15

Test Case ID	UT15
Relevant Test Objective	• Slave controller JSON control logic parser.
Test Steps	1. Develop valid control logic JSON files which conform to schema. 2. First control logic file contains sequential logic which contains single peripheral logic operations at each progression logic stage. Only discrete digital input/output values. 3. Second control logic file contains sequential logic which contains multiple peripheral logic operations at

	each progression logic stage. Only discrete digital input/output values. 4. Third control logic file contains sequential logic which contains single peripheral logic operations at each progression logic stage. Analog, discrete and encoded input/output parameter values. 5. Forth control logic file contains sequential logic which contains multiple peripheral logic operations at each progression logic stage. Analog, discrete and encoded input/output parameter values. 6. Fifth control logic file contains hybrid logic paths which contains multiple peripheral logic operations at each progression logic stage. Analog, discrete and encoded input/output parameter values. 7. Each control logic file will contain logical AND, OR and NOT logic operators. 8. Develop invalid control logic JSON files which do not conform to the schema. 9. First control logic file contains sequential logic which contains single peripheral logic operations at each progression logic stage. Only discrete digital input values. 10. Second control logic file contains sequential logic which contains multiple peripheral logic operations at each progression logic stage. Only discrete digital input values. 11. Third control logic file contains sequential logic which contains single peripheral logic operations at each progression logic stage. Analog, discrete and encoded input parameter values. 12. Forth control logic file contains sequential logic which contains multiple peripheral logic operations at each

	progression logic stage. Analog, discrete and encoded input parameter values. 13. Fifth control logic file contains hybrid logic paths which contains multiple peripheral logic operations at each progression logic stage. Analog, discrete and encoded input parameter values. 14. Each control logic file will contain logical AND, OR and NOT logic operators. 15. Implement and flash the slave controller JSON control logic parser sub-routine onto the master controller. 16. Load developed JSON control logic files onto SD card. 17. Load developed configuration file schemas onto SD card. 18. Mount SD card into master controller board and turn power on to the controller circuit. 19. HMI interface flashing indicates all control logic files parsed, log files stored to SD card and SD card unmounted from file system. 20. Power off master controller and eject SD card from controller. 21. Store generated JSON log files onto local personal computer for analysis.
Data Collected	• Sub-routine call log file: Records the sequence of sub-routines calls. • Control system error handling log file: Caught run-time errors. • Methodology test data log file: • Session management log file: • Control system evaluation log file: Outcome of logic expression evaluation. System context parameters at time of evaluation.

	• Unit test log files record internal data structures which store control logic parameters for each progression logic stage. • Unit test log files record logic expression constructed for each progression logic stage. • Unit test log files record system context at each progression logic stage. • Unit test log files record errors caused by invalid control logic files. • Unit test log files record run time errors.
Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file, control system evaluation log file matches expected outcome defined during implementation.

Table 14: Unit Test UT16

Test Case ID	UT16
Relevant Test Objective	• Slave controller JSON control logic interpreter.
Test Steps	1. Develop valid control logic JSON files which conform to schema. 2. First control logic file contains sequential logic which contains single peripheral logic operations at each progression logic stage. Only discrete digital input values. 3. Second control logic file contains sequential logic which contains multiple peripheral logic operations at each progression logic stage. Only discrete digital input values. 4. Third control logic file contains sequential logic which contains single peripheral logic operations at each

	progression logic stage. Analog, discrete and encoded input parameter values. 5. Forth control logic file contains sequential logic which contains multiple peripheral logic operations at each progression logic stage. Analog, discrete and encoded input parameter values. 6. Fifth control logic file contains hybrid logic paths which contains multiple peripheral logic operations at each progression logic stage. Analog, discrete and encoded input parameter values. 7. Each control logic file will contain logical AND, OR and NOT logic operators. Implement and flash slave control logic interpreter sub-routines onto the master controller. 8. Load developed JSON control logic files onto SD card. 9. Load developed configuration file schema onto SD card. 10. Mount SD card into slave controller board and turn power on to the controller circuit. 11. Unit test log files record run time errors. 12. HMI interface flashing indicates all control logic files parsed, log files stored to SD card and SD card unmounted from file system. Power off slave controller and eject SD card from controller. 13. Store generated JSON log files onto local personal computer for analysis.
Data Collected	• Sub-routine call log file: Records the sequence of sub-routines calls. • Control system error handling log file: Caught run-time errors. • Methodology test data log file:

	• Session management log file: • Control system evaluation log file: Outcome of logic expression evaluation. System context parameters at time of evaluation. • Unit test log files record logic expression constructed for each progression logic stage. • Unit test log files record system context at each progression logic stage. • Unit test log files record system context parameter values during interpreter evaluation. • Unit test log files record internal data structures which store control logic expressions and parameters for each progression logic stage. • Unit test log files record internal data structures which store control logic parameters for each progression logic stage. • Unit test log files record outcome of logic expression evaluation at each progression logic stage. • Unit test log files record errors caused by invalid control logic files.
Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file, control system evaluation log file matches expected outcome defined during implementation.

Table 15: Unit Test UT17

Test Case ID	UT17
Relevant Test Objective	• Slave controller JSON control logic state manager.
Test Steps	1. Develop valid control logic JSON files which conform to schema.

	2. First control logic file contains sequential logic which contains single peripheral logic operations at each progression logic stage. Only discrete digital input values. 3. Second control logic file contains sequential logic which contains multiple peripheral logic operations at each progression logic stage. Only discrete digital input values. 4. Third control logic file contains sequential logic which contains single peripheral logic operations at each progression logic stage. Analog, discrete and encoded input parameter values. 5. Forth control logic file contains sequential logic which contains multiple peripheral logic operations at each progression logic stage. Analog, discrete and encoded input parameter values. 6. Fifth control logic file contains hybrid logic paths which contains multiple peripheral logic operations at each progression logic stage. Analog, discrete and encoded input parameter values. 7. Each control logic file will contain logical AND, OR and NOT logic operators. 8. Implement and flash slave control logic state manager sub-routines onto the master controller. 9. Load developed JSON control logic files onto SD card. 10. Load developed configuration file schemas onto SD card. 11. Mount SD card into slave controller board and turn power on to the controller circuit. 12. HMI interface flashing indicates all control logic files parsed, log files stored to SD card and SD card unmounted from file system.

	13. Power off slave controller and eject SD card from controller. 14. Store generated JSON log files onto local personal computer for analysis.
Data Collected	• Sub-routine call log file: Records the sequence of sub-routines calls. • Control system error handling log file: Caught run-time errors. • Methodology test data log file: • Session management log file: • Control system evaluation log file: Outcome of logic expression evaluation. System context parameters at time of evaluation. • Unit test log files record system context of slave controller at each progression logic stage. • Unit test log files record system context parameter values of relevant slave controller peripherals for the current control logic progression stage. • Unit test log files record internal data structures which store control logic expressions and parameters for each progression logic stage. • Unit test log files record outcome of logic expression evaluation at each progression logic stage. • Unit test log files record system state and context transitions. • Unit test log files record errors caused by invalid control logic files. • Unit test log files record run time errors.
Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file, control system evaluation log file matches expected outcome defined during implementation.

Table 16: Unit Test UT18

Test Case ID	UT18
Relevant Test Objective	• Slave controller JSON control logic action dispatcher.
Test Steps	1. Load onto SD card the configuration file schema necessary for configuration. 2. Develop valid control logic JSON files which conform to master control logic schema. 3. First control logic file contains sequential logic which contains single peripheral logic operations at each progression logic stage. Only discrete digital input values. 4. Second control logic file contains sequential logic which contains multiple peripheral logic operations at each progression logic stage. Only discrete digital input values. 5. Third control logic file contains sequential logic which contains single peripheral logic operations at each progression logic stage. Analog, discrete and encoded input parameter values. 6. Forth control logic file contains sequential logic which contains multiple peripheral logic operations at each progression logic stage. Analog, discrete and encoded input parameter values. 7. Fifth control logic file contains hybrid logic paths which contains multiple peripheral logic operations at each progression logic stage. Analog, discrete and encoded input parameter values. 8. Each control logic file will contain logical AND, OR and NOT logic operators. 9. Distinct progression logic stages with different system context profiles.

	10. Implement and flash slave control logic action dispatcher sub-routines onto the slave controller. 11. Load developed JSON control logic files onto SD card. 12. Mount SD card into master controller board and turn power on to the controller. 13. HMI interface flashing indicates all control logic files parsed, log files stored to SD card and SD card unmounted from file system. 14. Store generated JSON file onto local personal computer for analysis.
Data Collected	• Sub-routine call log file: Records the sequence of sub-routines calls. • Control system error handling log file: Caught run-time errors. • Methodology test data log file: • Session management log file: • Control system evaluation log file: Outcome of logic expression evaluation. System context parameters at time of evaluation. • Unit test log files record sub-routine calls according to control system state and triggers. • Unit test log files record action dispatcher function calls. • Unit test log files records JSON files read from SD card. • Unit test log files record system state and context transitions. • Unit test log files record internal data structures generated from JSON file parsing. • Unit test log files record system context at each progression logic stage.

	• Unit test log files record system context parameter values during interpreter evaluation. • Unit test log files record internal data structures which store control logic expressions and parameters for each progression logic stage. • Unit test log files record outcome of logic expression evaluation at each progression logic stage. • Unit test log files record errors caused by invalid control logic files.
Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file, control system evaluation log file matches expected outcome defined during implementation.

Table 17: Unit Test UT19

Test Case ID	UT19
Relevant Test Objective	• Master controller communication action dispatcher.
Test Steps	1. Implement and load master_communication_action_dispatcher sub-routine for the slave controller. 2. Define command message schemas for master controller. 3. Load slave controller configuration files and mount SD card into SD card slot. 4. Power on the master controller. 5. Main function within slave controller simulates message commands and responses from master controller. 6. Power off and unmount SD card when HMI is flashing. 7. Store to local computer for analysis.

Data Collected	• Sub-routine call log file: Records the sequence of sub-routines calls. • Control system error handling log file: Caught run-time errors. • Methodology test data log file: • Session management log file: • Control system evaluation log file: Outcome of logic expression evaluation. System context parameters at time of evaluation.
Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file, control system evaluation log file matches expected outcome defined during implementation.

Table 18: Unit Test UT20

Test Case ID	UT20
Relevant Test Objective	• Slave controller communication action dispatcher.
Test Steps	1. Implement and load slave_communication_action_dispatcher sub-routine for the slave controller. 2. Define command message schemas for slave controller. 3. Load slave controller configuration files and mount SD card into SD card slot. 4. Power on the slave controller. 5. Main function within slave controller simulates message commands and responses from master controller. 6. Power off and unmount SD card when HMI is flashing. 7. Store to local computer for analysis.

Data Collected	• Sub-routine call log file: Records the sequence of sub-routines calls. • Control system error handling log file: Caught run-time errors. • Methodology test data log file: • Session management log file: • Control system evaluation log file: Outcome of logic expression evaluation. System context parameters at time of evaluation.
Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file, control system evaluation log file matches expected outcome defined during implementation.

Table 19: Unit Test UT21

Test Case ID	UT21
Relevant Test Objective	• Command message parser.
Test Steps	8. Implement and flash JSON command message parser subroutine onto the slave master and slave controllers. 9. Define JSON command message file schema on SD card storage. 10. Develop master to slave controller command message data packets which is structured from defined schema. 11. First command message JSON file is a command which queries the current logic stage evaluation request. Contains valid JSON syntax and structure. 12. Second command message JSON file is a command which queries the current logic stage evaluation request. Contains invalid JSON syntax and structure.

	13. Third command message JSON file is a command which sets current progression logic stage for target slave controller. Contains valid JSON syntax and structure. 14. Fourth command message JSON file is a command which sets current progression logic stage for target slave controller. Contains invalid JSON syntax and structure. 15. Develop slave Controller control logic file which is structured from defined schema. 16. Multiple scales of profiles. Also, some to have invalid JSON syntax and structure. 17. Develop slave Controller command message files which is structured from defined schema. 18. Multiple scales of profiles. Also, some to have invalid JSON syntax and structure. 19. Load master controller configuration file, slave control logic configuration file and slave communication profile file schemas to SD card. 20. Load developed JSON files onto SD card. 21. Mount SD card into slave controller board and turn power on to the controller. 22. HMI interface flashing indicates all configuration files parsed, log files stored to SD card and SD card unmounted from file system. 23. Store generated JSON file onto local personal computer for analysis.
Data Collected	• Sub-routine call log file: Records the sequence of sub-routines calls. • Control system error handling log file: Caught run-time errors. • Methodology test data log file: • Session management log file:

	• Control system evaluation log file: Outcome of logic expression evaluation. System context parameters at time of evaluation.
Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file, control system evaluation log file matches expected outcome defined during implementation.

Table 20: Unit Test UT23

Test Case ID	UT23
Relevant Test Objective	• Master controller command message interpreter.
Test Steps	1. Implement and flash command_message_interpreter routine onto the master controller. 2. Define valid command message schema for command message JSON file. 3. Develop command message UpdatePeripheral, ConfigStatus and UpdatePeripheral files which conform to the schema. 4. Format the SD card with the FAT32 file system. 5. Load master command message schema files and developed command message files onto SD card storage. 6. Define the expected outcome from generated log files. 7. Insert SD card storage into the master controller SD card slot and power on master controller board. 8. HMI interface flashing indicates all configuration files are parsed, validation is complete, log files are stored to the SD card, and the SD card is unmounted from the file system.

	9. Power off the master controller and eject SD card storage. 10. Store generated JSON file onto a local personal computer for document analysis.
Data Collected	• Command message schema artefact. • Unit23.txt stores the routines called in response to each command message.
Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file, control system evaluation log file matches expected outcome defined during implementation.

Table 21: Unit Test UT24

Test Case ID	UT24
Relevant Test Objective	• Slave controller command message interpreter.
Test Steps	1. Implement and flash command_message_interpreter routine onto the slave controller. 2. Define valid command message schema for command message JSON file. 3. Develop command message UpdatePeripheral, ConfigStatus and UpdatePeripheral files which conform to the schema. 4. Format the SD card with the FAT32 file system. 5. Load master command message schema files and developed command message files onto SD card storage. 6. Define the expected outcome from generated log files. 7. Insert SD card storage into the master controller SD card slot and power on master controller board.

	8. HMI interface flashing indicates all configuration files are parsed, validation is complete, log files are stored to the SD card, and the SD card is unmounted from the file system. 9. Power off the master controller and eject SD card storage. 10. Store generated JSON file onto a local personal computer for document analysis.
Data Collected	• Command message schema artefact. • Unit24.txt stores the routines called in response to each command message.
Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file, control system evaluation log file matches expected outcome defined during implementation.

3.2.2. Integration Testing

The tables within the integration testing section outline the integration tests which will be conducted to evaluate the outcomes and objectives of the escape room control system. Each integration test tests the interactions between integrated components to ensure the system components can communicate and respond to each other.

Table 22: Integration Test IT03

Test Case ID	IT03
Relevant Test Objective	Master controller communication action dispatcher and slave controller communication action dispatcher.
Test Steps	
Data Collected	• Sub-routine call log file: Records the sequence of sub-routines calls. • Control system error handling log file: Caught run-time errors. • Methodology test data log file:

	• Session management log file: • Control system evaluation log file: Outcome of logic expression evaluation. System context parameters at time of evaluation.
Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file, control system evaluation log file matches expected outcome defined during implementation.

Table 23: Integration Test IT04

Test Case ID	IT04
Relevant Test Objective	• Master controller initialisation and slave controller initialisation.
Test Steps	1. Develop and flash routine to ensure slave controller initalisation and before master controller completes initaisation. 2. Develop a configuration JSON file for both master controller and slave controller. 3. Develop a control_logic JSON file for both master controller and slave controller. 4. Format SD card with FAT32 file system. 5. Load configuration files onto SD card storage. 6. Define expected outcome from generated log files. 7. Power on master controller board. 8. Power on slave controller board. 9. HMI interface flashing indicates all configuration files parsed, log files stored to SD card and SD card unmounted from file system. 10. Power off master controller and eject SD card storage. 11. Power off slave controller and eject SD card storage.

	12. Store generated JSON file onto local personal computer for document analysis.
Data Collected	• Sub-routine call log file: Records the sequence of sub-routines calls. • Control system error handling log file: Caught run-time errors. • Control system evaluation log file: Outcome of logic expression evaluation. System context parameters at time of evaluation.
Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file, control system evaluation log file matches expected outcome defined during implementation.

Table 24: Integration Test IT05

Test Case ID	IT05
Relevant Test Objective	• Master controller initialisation and control logic engine
Test Steps	1. Develop and flash initalisation and control logic engine interfacing routines. 2. Develop a master controller configuration JSON file that is structured from a defined schema with correct syntax. 3. Develop a master controller control_logic JSON file that is structured from a defined schema with correct syntax. 4. Format SD card with FAT32 file system. 5. Load configuration files onto SD card storage. 6. Define expected outcome from generated log files. 7. Power on master controller board.

	8. HMI interface flashing indicates all configuration files parsed, log files stored to SD card and SD card unmounted from file system. 9. Power off slave controller and eject SD card storage. 10. Store generated JSON file onto local personal computer for document analysis.
Data Collected	• Sub-routine call log file: Records the sequence of sub-routines calls. • Control system error handling log file: Caught run-Control system evaluation log file: Outcome of logic expression evaluation. System context parameters at time of evaluation.
Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file, control system evaluation log file matches expected outcome defined during implementation.

Table 25: Integration Test IT06

Test Case ID	IT06
Relevant Test Objective	• Master controller control logic action dispatcher and communication action dispatcher.
Test Steps	1. Develop and flash control logic action dispatcher and communication engine action dispatcher interfacing routines. 2. Develop a master controller configuration JSON file that is structured from a defined schema with correct syntax. 3. Develop a master controller control_logic JSON file that is structured from a defined schema with correct syntax. 4. Format SD card with FAT32 file system. 5. Load configuration files onto SD card storage.

	6. Define expected outcome from generated log files. 7. Power on master controller board. 8. HMI interface flashing indicates all configuration files parsed, log files stored to SD card and SD card unmounted from file system. 9. Power off slave controller and eject SD card storage. 10. Store generated JSON file onto local personal computer for document analysis.
Data Collected	• Sub-routine call log file: Records the sequence of sub-routines calls. • Control system error handling log file: Caught run-time errors. • Control system evaluation log file: Outcome of logic expression evaluation. System context parameters at time of evaluation.
Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file, control system evaluation log file matches expected outcome defined during implementation.

Table 26: Integration Test IT07

Test Case ID	IT07
Relevant Test Objective	• Slave controller control logic action dispatcher and communication action dispatcher.
Test Steps	11. Develop and flash control logic action dispatcher and communication engine action dispatcher interfacing routines. 12. Develop a slave controller configuration JSON file that is structured from a defined schema with correct syntax.

	13. Develop a slave controller control_logic JSON file that is structured from a defined schema with correct syntax. 14. Format SD card with FAT32 file system. 15. Load configuration files onto SD card storage. 16. Define expected outcome from generated log files. 17. Power on master controller board. 18. HMI interface flashing indicates all configuration files parsed, log files stored to SD card and SD card unmounted from file system. 19. Power off slave controller and eject SD card storage. 20. Store generated JSON file onto local personal computer for document analysis.
Data Collected	• Sub-routine call log file: Records the sequence of sub-routines calls. • Control system error handling log file: Caught run-time errors. • Control system evaluation log file: Outcome of logic expression evaluation. System context parameters at time of evaluation.
Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file, control system evaluation log file matches expected outcome defined during implementation.

3.2.3. System Testing

The tables within the system testing section outline the system tests which will be conducted to evaluate the outcomes and objectives of the escape room control system. The system testing assesses the complete system control flow in simulated operational environment. This will verify the functional requirements of the system within escape room environments.

Table 27: System Test ST01

Test Case ID	ST01
Relevant Test Objective	• Operational escape room purpose test.
Test Steps	1. Design escape room narriative which utilising hybrid progression paths. 2. Develop control_logic.json and configuration.json files for master controller and three slave controller puzzles. 3. Format SD card with FAT32 file system. 4. Load slave controller configuration schema, master control logic schema and communication profile schema onto SD card storage. 5. Load slave controller configuration, control logic and command message JSON files onto SD card storage. 6. Define expected outcome from generated log files. 7. Format SD cards with FAT32 file system and insert each SD card into associated controller SD card slot. 8. Power on master controller board. 9. Power on slave controller boards. 10. Manually interact with the puzzle peripheals to traverse progression logic stages. 11. HMI interface flashing indicates all configuration files parsed, log files stored to SD card and SD card unmounted from file system. 12. Power off controllers and eject SD card storage. 13. Store generated JSON file onto local personal computer for document analysis.
Data Collected	• All unit and integration test log files will be enabled for logging.
Pass/Fail Criteria	Pass

	Sub-routine call log file, control system error handling log file, methodology call log file, session management log file, control system evaluation log file matches expected outcome defined during implementation.

Table 28: System Test ST02

Test Case ID	ST02
Relevant Test Objective	• Operational escape room user purpose test.
Test Steps	1. Design escape room narriative which utilising hybrid progression paths. 2. Develop control_logic.json and configuration.json files for master controller and three slave controller puzzles. 3. Format SD card with FAT32 file system. 4. Load slave controller configuration schema, master control logic schema and communication profile schema onto SD card storage. 5. Load slave controller configuration, control logic and command message JSON files onto SD card storage. 6. Define expected outcome from generated log files. 7. Format SD cards with FAT32 file system and insert each SD card into associated controller SD card slot. 8. Power on master controller board. 9. Power on slave controller boards. 10. Have users manually interact with the puzzle peripheals to traverse progression logic stages. 11. HMI interface flashing indicates all configuration files parsed, log files stored to SD card and SD card unmounted from file system. 12. Power off controllers and eject SD card storage.

	Store generated JSON file onto local personal computer for document analysis.
Data Collected	• All unit and integration test log files will be enabled for logging.
Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file, communication dump log file and control system evaluation log file matches expected outcome defined during implementation.

3.2.4. Performance Testing

The tables within the performance testing section outline the performance tests which will be conducted to evaluate the performance characteristics and points of failure for the outcomes and objectives. The performance tests will establish the limitations of the system through stress testing.

Table 29: Performance Test PT01

Test Case ID	PT01
Relevant Test Objective	• Operational communication coverage. • Evaluate and implement appropriate wireless communication methods, protocols and technologies for the master and slave controllers. • Investigate system usability limitations. Analysing system latency, data rate, loss tolerance, wireless communication distance and response time.
Test Steps	1. Design and flash master_communication_coverage_performance sub-routines onto the master controller device. 2. Design and flash slave_communication_coverage_performance sub-routines onto the slave controller device.

	3. Construct master controller configuration file, control logic file and command message files and load onto SD card storage. 4. Construct slave controller configuration file, control logic file and command message files and load onto SD card storage. 5. Mount the associated SD card storage into the master controller and slave controller devices. 6. Position the master controller in the fixed location behind the internal wall of a house as shown by. The master controller should be elevated off the floor by 1 meter. 7. Elevate the slave controller 1 meter from the floor at the first incremental location 1 meter distance from the master controller. 8. Power on the master controller device. 9. Power on the slave controller device. 10. Both controller devices will execute their communication coverage performance test routines. Controller HMI will flash once routines have completed. 11. If the slave controller distance from the master controller is less than 30 meters, increment the distance
Data Collected	• PT01.txt: RSSI, connection success, error rate of validated packets.
Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file, communication dump log file and control system evaluation log file matches expected outcome defined during implementation.

Table 30: Performance Test PT02

Test Case ID	PT02
Relevant Test Objective	• Operational communication latency.
Test Steps	1. Design and flash master_communication_latency_performance sub-routines onto the master controller device. 2. Design and flash slave_communication_ latency _performance sub-routines onto the slave controller device. 3. Construct master controller configuration file, control logic file and command message files and load onto SD card storage. 4. Construct slave controller configuration file, control logic file and command message files and load onto SD card storage. 5. Mount the associated SD card storage into the master controller and slave controller devices. 6. Position the master controller in the fixed location behind the internal wall of a house as shown by. The master controller should be elevated off the floor by 1 meter. 7. Elevate the slave controller 1 meter from the floor and 1 meter distance from the master controller. 8. Power on the master controller device. 9. Power on the slave controller device. 10. Both controller devices will execute their communication latency performance test routines. Controller HMI will flash once routines have completed. 11. Power off the master controller and eject SD card storage.

	12. Store generated JSON file onto a local personal computer for document analysis.
Data Collected	• Sub-routine call log file: Records the sequence of sub-routines calls. • Control system error handling log file: Caught run-time errors. • PT02.txt: Timer interrupt value of slave controller success message. • Control system evaluation log file: Outcome of logic expression evaluation. System context parameters at time of evaluation.
Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file, communication dump log file and control system evaluation log file matches expected outcome defined during implementation.

Table 31: Performance Test PT03

Test Case ID	PT03
Relevant Test Objective	• Concurrent slave controller communication to master controller performance test.
Test Steps	1. Design and flash master_concurrent_comm_performance sub-routines onto the master controller device. 2. Design and flash slave_concurrent_comm_performance sub-routines onto the slave controller device. 3. Construct master controller configuration file, control logic file and command message files and load onto SD card storage.

	4. Construct slave controller configuration file, control logic file and command message files and load onto SD card storage. 5. Mount the associated SD card storage into the master controller and slave controller devices. 6. Position the master controller in the fixed location behind the internal wall of a house as shown by. The master controller should be elevated off the floor by one meter. 7. Elevate the each slave controllers one meter from the floor and one meter distance from the master controller. 8. Power on the master controller device. 9. Power on the slave controller devices. 10. All controller devices will execute their communication concurrency performance test routines. Controller HMI will flash once routines have completed. 11. Power off the master controller and eject SD card storage. 12. Store generated JSON file onto a local personal computer for document analysis.
Data Collected	• Sub-routine call log file: Records the sequence of sub-routines calls. • Control system error handling log file: Caught run-time errors. • Control system evaluation log file: Outcome of logic expression evaluation. System context parameters at time of evaluation. • PT03.txt: Concurrent connections and command message processing time for local device. • Communication Dump: The command message files received and sent by device.

Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file, communication dump log file and control system evaluation log file matches expected outcome defined during implementation.

Table 32: Performance Test PT04

Test Case ID	PT04
Relevant Test Objective	• Concurrent control logic evaluation performance test.
Test Steps	1. Design and flash concurrent_logic_performance sub-routines onto the master controller device. 2. Design and flash concurrent_logic_performance sub-routines onto the slave controller device. 3. Construct master controller configuration file, control logic file and command message files and load onto SD card storage. 4. Construct slave controller configuration file, control logic file and command message files and load onto SD card storage. 5. Mount the associated SD card storage into the master controller and slave controller devices. 6. Position the master controller in the fixed location behind the internal wall of a house as shown by. The master controller should be elevated off the floor by one meter. 7. Elevate the each slave controller one meter from the floor and one meter distance from the master controller. 8. Power on the master controller device. 9. Power on the slave controller devices.

	10. All controller devices will execute their control logic progression stages through concurrent_performance test routines. Controller HMI will flash once routines have completed. 11. Power off the master controller and eject SD card storage. 12. Store generated JSON file onto a local personal computer for document analysis.
Data Collected	• Sub-routine call log file: Records the sequence of sub-routines calls. • Control system error handling log file: Caught run-time errors. • PT04: Evaluation time of each progression logic stage. • Control system evaluation log file: Outcome of logic expression evaluation. System context parameters at time of evaluation.
Pass/Fail Criteria	Pass Sub-routine call log file, control system error handling log file, methodology call log file, session management log file, communication dump log file and control system evaluation log file matches expected outcome defined during implementation.

CHAPTER 4: RESULTS

4.1. Testing Process

The testing process followed the sequential development stages according to the Development Process section of methodology. The following is the final implementation of these development stages and the resulting document artefacts from the associated test cases.

4.1. Embedded Architecture of Master Controller and Slave Controller

The embedded controller design within 3.1.1 of the methodology describes the specification for interface electronics of both the master controller and slave controllers. The master controller requires that Bluetooth BLE, power regulation, clocking circuitry, SD card port and LED indicator hardware be included within the embedded controller design. As outlined within Table 1: Unit Test UT01, the master controller's embedded system was designed and assembled according to Table 33: Master controller ESP32-C6-DevKit-1 v1.2 pin assignment.

Table 33: Master controller ESP32-C6-DevKit-1 v1.2 pin assignment

Pin Description	GPIO Number	Header.Pin	Notes
MOSI	GPIO4	J1.3	SD Card SPI
CLK	GPIO5	J1.4	SD Card SPI
MISO	GPIO6	J1.5	SD Card SPI
CS	GPIO7	J1.6	SD Card SPI
RGB LED	GPIO8	J1.9	Built into DevKit
5V	5V	J1.14	5V->3.3V Voltage Regulator
GND	GND	J1.15	Ground

The slave controller's specification requires the addition of a peripheral microcontroller input/output. The SPI interface between an ATMEGA328P and the ESP32-C6-DevKitC-1 v1.2 failed due to the input/output reference logic levels being different for their SPI interface. An intermediate level shifting integrated circuit was used to address the communication issues. However due to time containts the remianing input and output peripherals of the slave controller were allocated as peripherals for the project.

The pin allocation for the slave controller is shown in Table 34: Slave controller ESP32-C6-DevKit-1 v1.2 pin assignment. The strapping pins of the ESP32-C6 weren't allocated as available input and output peripherals to avoid configuration errors during power up.

Table 34: Slave controller ESP32-C6-DevKit-1 v1.2 pin assignment

Pin Description	GPIO Number	Header.Pin	Notes
MOSI	GPIO4	J1.3	SD Card SPI
CLK	GPIO5	J1.4	SD Card SPI
MISO	GPIO6	J1.5	SD Card SPI
CS	GPIO7	J1.6	SD Card SPI
RGB LED	GPIO8	J1.9	Built into DevKit
5V	5V	J1.14	5V->3.3V Voltage Regulator
GND	GND	J1.15	Ground
Peripheral	GPIO0	J1.7	ADC1_CH0, GPIO
Peripheral	GPIO1	J1.8	ADC1_CH1, GPIO
Peripheral	GPIO2	J1.12	ADC1_CH2, GPIO
Peripheral	GPIO3	J1.13	ADC1_CH3, GPIO
Peripheral	GPIO10	J1.10	GPIO
Peripheral	GPIO11	J1.11	GPIO
Peripheral	GPIO12	J3.14	GPIO
Peripheral	GPIO13	J3.13	GPIO
Peripheral	GPIO22	J3.6	GPIO
Peripheral	GPIO23	J3.	GPIO

The sd_card_interface routine is required to execute read, write, append, mount and unmounting SD card operations. In order to achieve the these requirements the helper functions sd_card_init, sd_card_write_file, sd_card_append_file, sd_card_read_file_dynamic and sd_card_deinit were developed. The sd card functions were developed by referencing ESP-IDF example documentation and then customised for the control system's specific use case. The final source code

implementation for the sd_card_interface routine can be found in UT01 - Embedded Architecture of Master Controller.

The sd_card_init header definition is shown in Source Code Snippet 1: sd_card_init. The sd_card_init function initalises the SPI bus and configures the SD card for communication. The FAT filesystem is mounted which enables the file operations. If either stage fails, the function will deinitalise the SPI bus and return an error.

```
// Initialise and mount the SD card filesystem.
esp_err_t sd_card_init(void);
```

Source Code Snippet 1: sd_card_init

The sd_card_write_file function creates the file at the specified path if it doesn't exist. Once the file is created, it opens the file in write mode and writes the data parameter to the file. The file is then closed after writing with an error returned if the file wasn't completely wrtten. The sd_card_write_file function definition is shown in Source Code Snippet 2: sd_card_write_file.

```
// Write string data to path.
esp_err_t sd_card_write_file(const char *path, const char *data);
```

Source Code Snippet 2: sd_card_write_file

The sd_card_append_file function appends a null terminated string to the path specified by the path parameter. The function opens the file in append mode and then adds the new data to the end of the file. The function will return an error or success code depending on the operations outcome. The sd_card_append_file definition is shown in Source Code Snippet 3: sd_card_append_file.

```
// Append string data to path.
esp_err_t sd_card_append_file(const char *path, const char *data);
```

Source Code Snippet 3: sd_card_append_file

The sd_card_read_file_dynamic function reads the entire content of the specified file into a dynamically allocated buffer. The function determines the file size, allocates the memory to fit the file content and then reads the data from the SD card. The calling function is then responsible for releasing the buffer memory resources once no longer required. The sd_card_read_file_dynamic definition is shown in Source Code Snippet 4: sd_card_read_file_dynamic.

```
// Read file from path. Dynamic allocation in memory for dynamic file sizes.
esp_err_t sd_card_read_file_dynamic(const char *path, char **buffer, size_t
*file_size);
```

Source Code Snippet 4: sd_card_read_file_dynamic

The sd_card_deinit function unmounts the FAT file system and then releases the SPI bus resources to ensure data is safely written before system shutdown. The sd_card_deinit definition is shown in Source Code Snippet 5: sd_card_deinit.

```
// Unmount the SD card filesystem and deinitialise
void sd_card_deinit(void);
```

Source Code Snippet 5: sd_card_deinit

The test procedure for sd_card_interface routine is outlined within Table 1: Unit Test UT01. The file content shown within unit01_read.txt was successfully appended to the unit01.txt log file. This validates that the sd_card_read_dynamic function successfully retireved the content of unit01_read.txt. SD card initalisation is validated through both files being operated on by the file system expressions. The resulting content within unit01.txt log file is shown in Log Artefact 1: unit01.txt. The unit test artefacts and source code implementation specific to UT01 are shown within in the respective sections of UT01 - Embedded Architecture of Master Controller. Refer to Table 35: UT01 outcome matrix for unit test results.

sd_card_init success.
Content of unit01_read.txt
sd_card_append_file success.

Log Artefact 1: unit01.txt

Table 35: UT01 outcome matrix

Test Case	Test Description	Outcome
Stage 01	SD card init function initalises SPI bus and mounts file system.	Pass
Stage 02	SD card write function overwrites specified file content and creates file with data.	Pass
Stage 03	SD card append file function appends the data to the end of specified file contet.	Pass
Stage 04	SD card read function reads dynamic file size to memory buffer.	Pass

Stage 05	SD card deinit function deinitalises SPI bus and unmounts file system.	Pass

The slave controller embedded architecture utilises the same sd_card_interface routine as the master controller. An additional embedded routine is required for the slave controller which is peripheral_update. The peripheral_update routine was implemented with the function definition shown in Source Code Snippet 6: peripheral_update. The process flow diagram for the peripheral_update routine is shown in Figure 5: peripheral_update process flow.

```
// Read or Write to Peripheral GPIO
esp_err_t peripheral_update(UpdateType update_type);
```

Source Code Snippet 6: peripheral_update

The peripheral_update routine provides a data structure interface for other routines to access and update the value of specific peripherals. These data structures are shown in the Source Code Implementation section of UT02 - Embedded Architecture of Slave Controller. The primary type definition is the PinPeripheral which is accessed by peripheral_update through it's PinDirection property. Other control system routines such as the control logic engine's action dispatcher and state manage will reference the PinPeripheral instances through the PeripheralID property key. The PinPeripheral data structure type definition is shown in Source Code Snippet 7: type definition of PinPeripheral.

```
typedef struct PinPeripheral {
    char* PeripheralID;
    int GPIONumber;
    PinDirection PinDirection;
    PinSignalType PinSignalType;
    ValueType PeripheralDataType;
    JsonValue* PeripheralValue;
} PinPeripheral;
```

Source Code Snippet 7: type definition of PinPeripheral

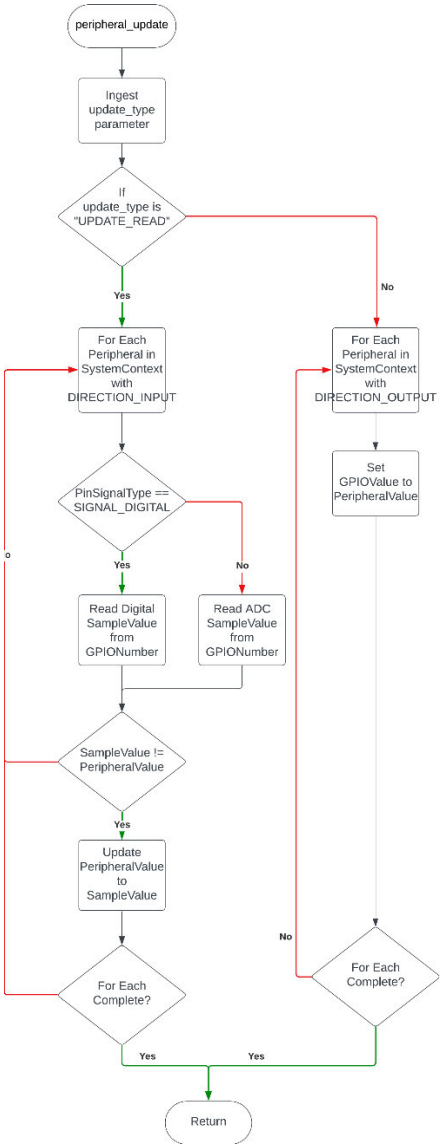


Figure 4: peripheral_update process flow

The test procedure for update_peripheral routine is outlined within Table 2: Unit Test UT02. The file content shown within unit02.txt demonstrates the routines ability to execute both read and write operations to the peripheral GPIO pins configured within the system context array. The main source code file process diagram for the unit test is shown in UT02 - Embedded Architecture of Slave Controller.

Unit Test 02 - Start
Read: 2 = 124
Read: 10 = true
Write: 22 = false

Log Artefact 2: unit02.txt

4.2. Embedded JSON Serialiser

As outlined in methodology section Embedded JSON Serialiser, the embedded JSON serialiser requires that json_schema_serialiser, get_schema_content routines, and the JSON schema file be developed to achieve its specifications. As the generated JSON file can contain multiple data types within property values, the valid data types were defined within the header file shown in UT03 – Embedded JSON Serialiser. As seen in Source Code Snippet 8: ValueType definition, the ValueType enumeration defines the valid data types that can be assigned to JSON properties or internal data structures. Source Code Snippet 9: JsonValueUnion definition shows the type definition of JsonValueUnion, which stores the JSON content's value depending on the selected ValueType within the JsonValue structure.

```
typedef enum {
    TYPE_INVALID,
    TYPE_INT,
    TYPE_FLOAT,
    TYPE_STRING,
    TYPE_BOOL,
    TYPE_CHAR
} ValueType;
```

Source Code Snippet 8: ValueType definition

```
typedef union {
    int int_val;
    float float_val;
    const char* str_val;
    bool bool_val;
    char char_val;
} JsonValueUnion;
```

Source Code Snippet 9: JsonValueUnion definition

The JsonValue structure type definition then contains the ValueType and JsonValueUnion for a given JSON content literal. This type definition enables all routines to have a single, flexible data type for processing various JSON content values. ValueType declares the data type, and the value is stored within the JsonValueUnion union. The JsonValue type definition is shown in Source Code Snippet 10: JsonValue definition

```
typedef struct {
    ValueType type;
    JsonValueUnion value;
} JsonValue;
```

Source Code Snippet 10: JsonValue definition

The primary functionality of the json_schema_serialiser routine is to ingest a target JSON schema name, read the schema from SD card storage, traverse it, and generate a JSON file conforming to it during the process. The json_schema_seraliser can be called using the function prototype shown in Source Code Snippet 11: json_schema_seraliser function prototype. The process diagram in Figure 5: json_schema_serialiser flow diagram shows the routine's progression.

```
char* json_schema_serialiser(const char* schema_name);
```

Source Code Snippet 11: json_schema_seraliser function prototype

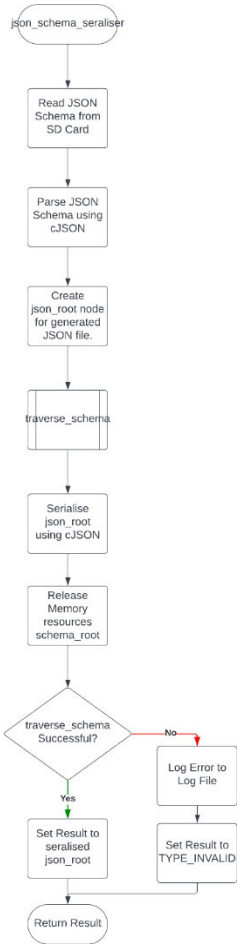


Figure 5: json_schema_serialiser flow diagram

The json_schema_seraliser routine has two helper functions: get_terminal_value and traverse_schema. The traverse_schema helper function is recursive and traverses the schema's nested structure. Then, depending on the type of property the function encounters, it will generate the corresponding JSON string and add it to the root_json object. The root_json object is the starting object from which the resulting JSON file will be constructed. The base case of the recursive function is when only terminal properties exist within either an object or array JSON property. When a terminal

property is encountered within the schema, the get_terminal_value is called to retrieve the data content from the correct internal data structure. The get_terminal_value ingests the schema_name, property_name and object_index parameters to call the get_schema_content routine. The get_terminal_value retrieves the terminal property's value from internal data structures and returns the content within the associated JSON string. Once the terminal's JSON string has been returned to traverse_schema, the terminal property's string is added to the root_json object.

```
// Function Prototypes
static cJSON* get_terminal_value(const char* schema_name, const char*
property_name, int object_index);
static bool traverse_schema(cJSON* schema_node, const char* schema_name,
cJSON* json_node, int object_index);
```

Source Code Snippet 12: json_schema_serialiser helper function prototypes

The get_schema_content routine uses SchemaPropertyMapping structures, which map schema name, property name and object index tuple to a retrieval function. This retrieval allows O(n) retrieval of content values specific to the JSON file being generated. The retrieval functions are organised into sub-routine C files by schema name, with the header files being included within the get_schema_content C file. This organisation of retrieval functions keeps the get_schema_content routine to a maintainable file length. The SchemaPropertyMapping is shown in Figure 7: SchemaPropertyMapping type definition.

```
// Function to retrieve content based on schema and property
JsonValue get_schema_content(const char* schema_name, const char*
target_property, int object_index);
```

Figure 6: get_schema_content function prototype

```
// Struct for mapping schema and property to retrieval functions
typedef struct {
    const char* schema_name;
    const char* property_name;
    RetrievalFunction function;
} SchemaPropertyMapping;
```

Figure 7: SchemaPropertyMapping type definition

Table 38: UT03 Outcome Matrix captures the results of each test case for Table 3: Unit Test UT03. The resulting test artefacts from the unit test cases can be found in UT03 – Embedded JSON Serialiser.

Table 36: UT03 Outcome Matrix

Test Case	Test Description	Outcome
Schema1.json	Validate data type of only terminal properties.	Pass
Schema2.json	Validate object and terminal properties at same nested level.	Pass
Schema3.json	Validate deeply nested objects.	Pass
Schema4.json	Validate deeply nested arrays and objects.	Pass

The embedded JSON serialiser and its components are validated according to Table 3: Unit Test UT03, which outlines that multiple JSON schema files are to be developed to ensure that varying JSON file structures can be generated. The unit test generated five differing JSON files using five different JSON schemas shown in UT03 – Embedded JSON Serialiser validated the json_schema_seraliser routine's ability to generate JSON files containing terminal properties of each data type. The test can be evaluated as successful when comparing the resulting schema1.json file content below against the schema definition. The JSON file contains the correct property names, with the data retrieved by get_schema_content matching the expected data type and value.

Schema01.json

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Schema1",
  "type": "object",
  "properties": {
    "string_prop": {
      "type": "string",
      "description": "A string property."
    },
    "int_prop": {
      "type": "integer",
      "description": "An integer property."
    },
    "float_prop": {
      "type": "number",
      "description": "A floating-point number property."
```

```
    },
    "bool_prop": {
      "type": "boolean",
      "description": "A boolean property."
    }
  },
  "required": ["string_prop", "int_prop", "float_prop", "bool_prop"],
  "additionalProperties": false
}
```

Log Artefact 3: schema1.json

```
{
    "string_prop": "String Value",
    "int_prop": 1997,
    "float_prop": 19.950000762939453,
    "bool_prop": true
}
```

Log Artefact 4: schema1_output.json

The schema2.json schema file validates the json_schema_seraliser routine's ability to generate JSON files that contain objects and terminal properties at the same nested level. The test can be evaluated as successful as the property names of the JSON file align with the schema2.json schema definition. The data contained by each property also matches the data type and value of the associated retrieval function's data structure content.

```
{
    "parent_prop": {
        "child_string_prop": "Nested String Value",
        "child_int_prop": 123
    },
    "main_float_prop": 45.669998168945312,
    "main_bool_prop": true
}
```

Log Artefact 5: schema2_output.json

The schema3.json schema file validated the json_schema_seraliser routines ability to generate JSON files that had deeply nested objects and terminal properties.

```
{
    "level1_prop": {
        "level1_string_prop": "Level 1 String",
        "level2_prop": {
            "level2_int_prop": 2023,
            "level3_prop": {
                "level3_bool_prop": true,
```

```
                "level4_prop": {
                    "level4_string_prop": "Deeply Nested String",
                    "level4_int_prop": 42
                },
                "level3_float_prop": 123.45600128173828
            }
        }
    },
    "root_bool_prop": true
}
```


Log Artefact 6: schema3_output.json

The schema4.json schema file validated the json_schema_seraliser routine ability to generate JSON files, which contained arrays with multiple elements, objects and terminal properties that contained different content values.


```
{
    "Controllers": [
        {
            "string_prop": "Controller1_String",
            "int_prop": 100,
            "float_prop": 123.44999694824219,
            "bool_prop": true
        },
        {
            "string_prop": "Controller2_String",
            "int_prop": 200,
            "float_prop": 678.9000244140625,
            "bool_prop": false
        }
    ]
}
```


Log Artefact 7: schema4_output.json

The schema5.json schema file validated the json_schema_seraliser routine ability to generate JSON files that contain nested arrays, objects and terminal properties.


```
{
    "UnitTests": [
        {
            "unit_test_profile": {
                "unit_test_name": "UnitTest1",
                "unit_test_ID": 101,
                "unit_test_state": true
            },
            "unit_test_controller": {
                "controller_role": "Master",
                "controller_ID": 201
```

```
            },
            "unit_test_log_files": [
                {
                    "log_file_name": "LogFile1",
                    "log_file_version": 1.1000000238418579,
                    "sub_routine": {
                        "parameter": "ParameterA",
                        "parameter_value": 301
                    }
                },
                {
                    "log_file_name": "LogFile2",
                    "log_file_version": 2.2000000476837158,
                    "sub_routine": {
                        "parameter": "ParameterB",
                        "parameter_value": 302
                    }
                }
            ]
        },
        {
            "unit_test_profile": {
                "unit_test_name": "UnitTest2",
                "unit_test_ID": 102,
                "unit_test_state": false
            },
            "unit_test_controller": {
                "controller_role": "Slave",
                "controller_ID": 202
            },
            "unit_test_log_files": [
                {
                    "log_file_name": "LogFile1",
                    "log_file_version": 1.1000000238418579,
                    "sub_routine": {
                        "parameter": "ParameterA",
                        "parameter_value": 301
                    }
                }
            ]
        }
    ]
}
```


Log Artefact 8: schema5_output.json

The evaluation of each unit test for the embedded JSON serialiser demonstrates the routine's ability to generate flexible JSON file structures that conform to a target

schema. As outlined by the Pass/Fail criteria within Table 3: Unit Test UT03, the generated JSON file matches the structure and content within the associated JSON schema and the generated JSON files were stored in the mounted SD card storage. Therefore, the embedded JSON schema serialiser operates in isolation according to its unit test specification.

4.3. Bluetooth BLE Connection Interface

The Bluetooth BLE interface was implemented using the nimBLE ESP-IDF compoents by Espressif Technologies. The master controller was configured as a BLE central device, with the slave controllers being set as BLE peripherals. This enabled multiple bi-directional connections between the master controller and slave controllers to be managed. The examples at the below ESP-IDF directories were implemented for the BLE interface and provided the necessary functions.

```
${IDF_PATH}/examples/bluetooth/nimble/ble_multi_conn/ble_multi_conn_cent
${IDF_PATH}/examples/bluetooth/nimble/blecent
${IDF_PATH}/examples/bluetooth/nimble/bleprph
```

The example source files make use of the nimBLE framework component provided by the ESP-IDF v5.3.1 found and were added to the project by including the below component into the CmakeList.txt file.

```
${IDF_PATH}/components/bt
```

Table 4: Unit Test UT04 outlines the test cases in which the Bluetooth BLE interface must pass in order to meet the objectives and outcome of a robust communication method tailored for master-slave data transfer. The resulting outcome from each test case in the unit test is shown in Table 37: UT04 Outcome Matrix.

Table 37: UT04 Outcome Matrix

Test Case	Test Description	Outcome
1	Master controller establishes communication with one slave controller.	Pass
2	Master controller transfer JSON file to slave controller 01.	Pass

3	Master controller handles multiple BLE connections.	Pass
4	Master controller transfers JSON file to specific slave controller 02 with multiple connections.	Pass
5	Master controller disconnects from specific slave controller 02 while managing multiple connections.	Pass

The resulting CommunicationDump.txt log file for the master controller is shown in Log Artefact 9: Master BLE CommunicationDump.txt. This log file captures the communication events which occurred during the unit test and demonstrates that the master controllres ability to reliably manage multiple connections with slave controllres.

```
MasterController – CommunicationDump.txt
Connect to SlaveController01
Connection Established with SlaveController01
Transfer json_message01.json to SlaveController01
Connect to SlaveController02
ConnectionEstablished with SlaveController02
Connect to SlaveController03
ConnectionEstablished with SlaveController03
Disconnect SlaveController01
Transfer json_message02.json to SlaveController02
Transfer json_message03.json to SlaveController03
Disconnect SlaveController02
Disconnect SlaveController03
```

Log Artefact 9: Master BLE CommunicationDump.txt

The resulting CommunicationDump.txt log file for each slave controller in the unit test are shown in Log Artefact 10: Slave01 BLE Communication Dump.txt, Log Artefact 11: Slave02 BLE CommunicationDump.txt and Log Artefact 12: Slave03 CommunicationDump.txt. The log file for each slave controller captures the communication events which occurred during the unit test and demonstrates that the slave controllres ability to reliably respond to master controller BLE interface events.

```
SlaveController01 - CommunicationDump.txt
Connection Request with Master
Connection Established with Master
Transfer ack_payload.json to Master
MessagePayload: { "Message": "SlaveCtrl01" }
Disconnect Master
```

Log Artefact 10: Slave01 BLE Communication Dump.txt

```
SlaveController02 - CommunicationDump.txt
Connection Request with Master
Connection Established with Master
Transfer ack_payload.json to Master
MessagePayload: { "Message": "SlaveCtrl02" }
Disconnect Master
```

Log Artefact 11: Slave02 BLE CommunicationDump.txt

```
SlaveController03 - CommunicationDump.txt
Connection Request with Master
Connection Established with Master
Transfer ack_payload.json to Master
MessagePayload: { "Message": "SlaveCtrl03" }
Disconnect Master
```

Log Artefact 12: Slave03 CommunicationDump.txt

The evaluation of each unit test for the Bluetooth BLE interface demonstrates the routine's ability to manage multiple BLE connection instances and allow bi-directional transfer of JSON files. As outlined by the Pass/Fail criteria within Table 4: Unit Test UT04, the Bluetooth BLE interface operates as expected in isolation according to it's unit test specifications.

4.4. Master Controller Configuration

The master controller configuration engine is responsible for initalising the master controller's system context and ensuring the slave controller's have all successfully been configured before the first progression logic stage is loaded. Table 5: Unit Test UT05 outlines the test cases in which the master configuration parser must pass in order to meet the objectives and outcome of serperation of hardware and software implementation through configuraiton files and a sclable internal representation session system context profiles. The resulting outcome from each test case in the unit test is shown in Table 38: UT05 outcome matrix.

Table 38: UT05 outcome matrix

Test Case	Test Description	Outcome
Stage 01	Creation of system context strucutres.	Pass
Stage 02	Configuration file loaded from SD card.	Pass
Stage 03	Device profiles constructed from master_config.json	Pass

Table 6: Unit Test UT07 outlines the test cases in which the master configuration interpreter must pass in order to meet the objectives and outcome of serperation of hardware and software implementation through configuraiton files and a sclable internal representation session system context profiles. The resulting outcome from each test case in the unit test is shown in Table 39: UT07 outcome matrix.

Table 39: UT07 outcome matrix

Test Case	Test Description	Outcome
Stage 01	The master_config.json parsed into DeviceProfiles system context. ConfigurationStatus set to true for all DeviceProfiles.	Pass
Stage 02	Each DeviceProfile within DeviceProfiles has DeviceProfile.ConfigurationStatus checked if true.	Pass
Stage 03	Outcome of configuration check returned true	Pass

4.5. Slave Controller Configuration

Table 7: Unit Test UT08 outlines the test cases in which the slave controller's configuration parser must pass in order to meet the objectives and outcomes. The resulting outcome from each test case in the unit test is shown in Table 40: UT08 outcome matrix.

Table 40: UT08 outcome matrix

Test Case	Test Description	Outcome
Stage 01	Creation of system context strucutres.	Pass
Stage 02	Configuration file loaded from SD card.	Pass
Stage 03	The slave_config.json parsed into DeviceProfiles and PinPeripherals system context structures.	Pass

Table 8: Unit Test UT10 outlines the test cases in which the slave configuration interpreter must pass in order to meet the objectives and outcomes. The resulting outcome from each test case in the unit test is shown in Table 41: UT10 outcome matrix.

Table 41: UT10 outcome matrix

Test Case	Test Description	Outcome
Stage 01	The slave_config.json parsed into DeviceProfiles and PinPeripherals system context.	Pass
Stage 02	update_peripheral routine initalises PinPeripheral.PeripheralValue for each in PinPeripherals	Pass
Stage 03	Outcome of configuration check returned true	Pass
Stage 04	DeviceProfile.ConfigurationStatus set to outcome	Pass

4.6. Master Controller's Control Logic Engine

The master controller's control logic engine is responsible for evaluating control logic expression outcomes for all slave controller devices relevant to the current progression logic stage. The control_logic_parser routine is responsible for parsing the control_logic_config.json file into the internal data structures for efficeient management of the current progression logic stage. The function definition for the master controller's control_logic_parser is shown in Source Code Snippet 14: master control_logic_parser prototype. The control_logic_parser ingests a pointer refereencing the root cJSON representation of the current progression logic stage.

```
// Master control logic parser prototype definition.
bool control_logic_parser(cJSON* json_root);
```

Source Code Snippet 13: master control_logic_parser prototype

The control logic parser then follows the abstracted process diagram outlined in Figure 8: master control logic parser abstracted process diagram. The complete source code for control_logic_parser routine is found in UT11 - Master Controller's Control Logic Engine Parser.

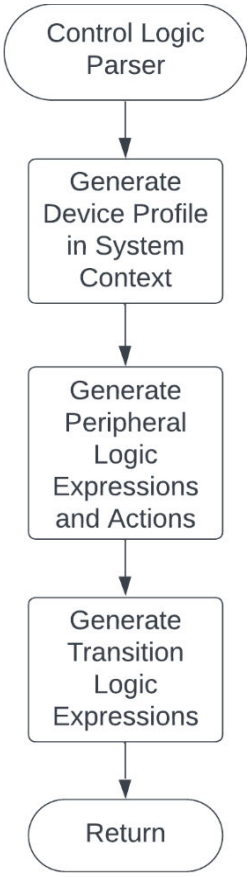


Figure 8: master control logic parser abstracted process diagram

The control_logic_parser routine has two additional helper routines; traverse_expression and create_instruction_stack. The traverse_expression routine recuresivly traverses the nested JSON expression structure and constructs the an expression tree from ExpressionNode and ExpressionOperand structures. This is shown in Figure 9: traverse_expression process diagram.

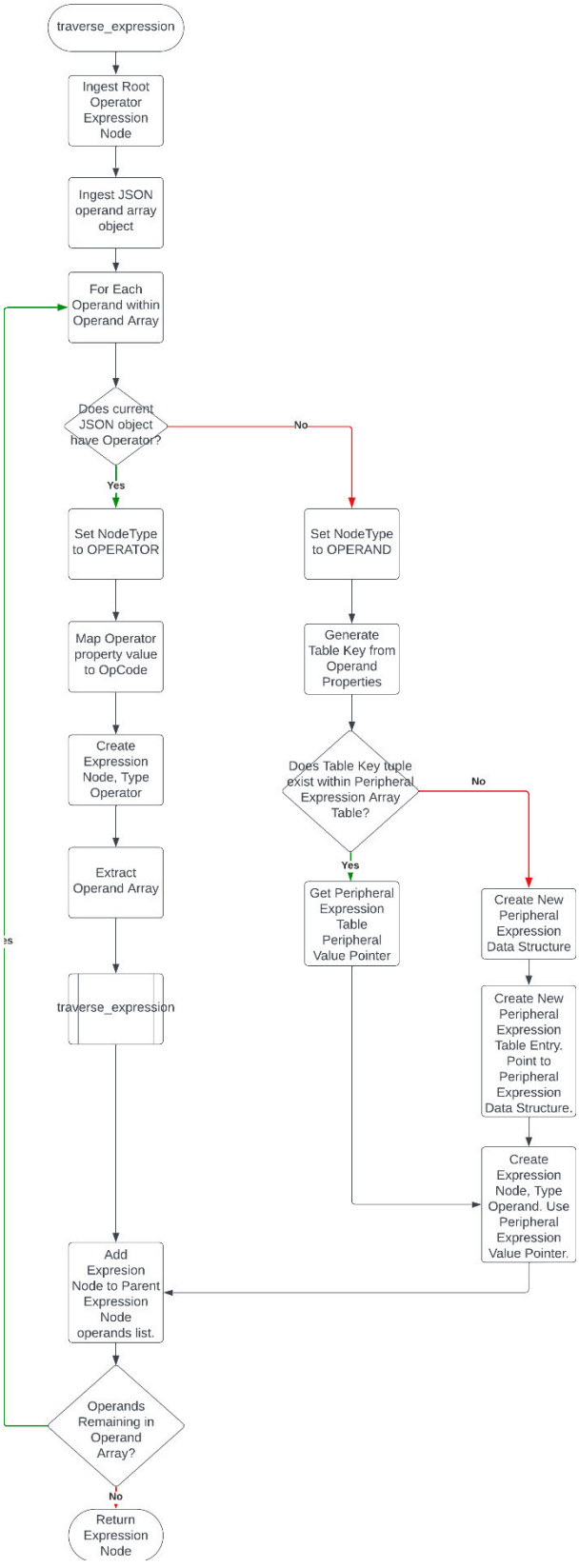


Figure 9: traverse_expression process diagram

Both ExpressionNode and ExpressionOperand are shown within control_logic_parser.h in UT11 - Master Controller's Control Logic Engine Parser. The expression tree is constructed from nested ExpressionNode structures which have a

disjoint union between it's two types of either NODE_OPERATOR or NODE_OPERAND. The relational model of this structure is shown in Figure 10: ExpressionNode relationship to sub-types. The design decision to declare OperandData and OperatorData subtype definitions within union of ExpressionNode. Reduces memory fragmentation, footprint and management for each ExpressionNode. The OperandData and OperatorData structures only exist during the lifecycle of the ExpressionNode. Therefore, they can their memory allocation can be released when the associated ExpressionNode is also released.

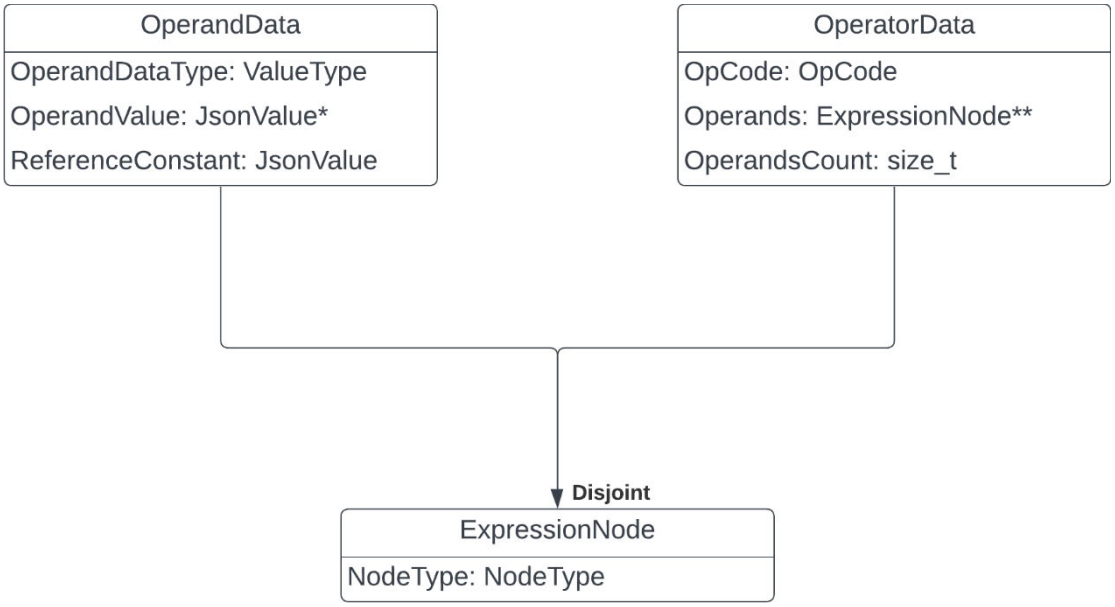


Figure 10: ExpressionNode relationship to sub-types

The OperandValue within the OperandData sub-type is a pointer to the pointer value of the ExpressionOperand structure's OperandValue propoerty. This decision was made since the memory resources of ExpressionNode structures are released once the RPN instruction stack is generated by the create_intstruction_stack helper function. The OperandData passes the memory location of the ExpressionOperand's OperandValue property so that the generated instruction operand value persits beyond the lifecycle of the ExpressionNode. The create_instruction_stack helper function implements post-order traversal to generate an array of Instruction structures which represent the expression tree in reverse polish notation. This keeps the nested JSON expressions precedence and associativity and enables the Instruction stack to be sequentally processed to evaluate the LogicExpression structure. The ExpressionNode parses the memory location of the OperandValue property in the

ExpressionOperand structure. This allows the control_logic_interpreter to directly reference the value within the relevant expression operand structure. The ExpressionOperand structure is then updated by the control logic state manager when operand changes occur within the slave controllers. The relationship diagram in Figure 11: expression structure relationships shows the entity definitions of each of these structures.

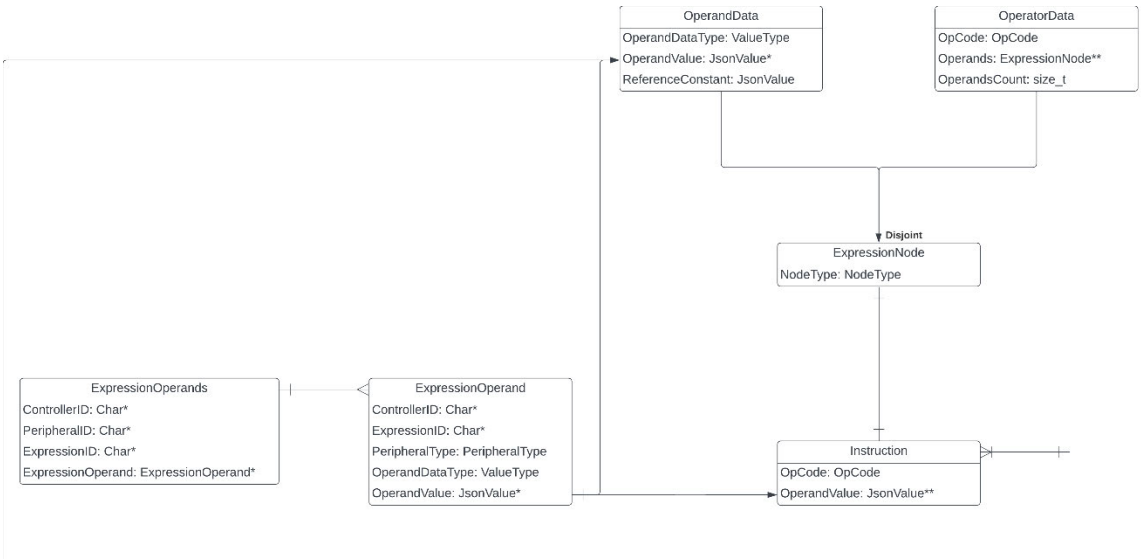


Figure 11: expression structure relationships

Once the nested JSON expression has been parsed into it's LogicExpression structure, it is then contained within either a PeripheralLogicExpression or TransitionLogicExpression structure. As shown within Figure 8: master control logic parser abstracted process diagram, the PeripheralLogicExpression structures are generated before the TransitionLogicExpression structures as the outcome of the PeripheralLogicExpression structures become the ExpressionOperands for the TransitionLogicExpression instructions. The control_logic_parser routine constructs an array of Action structures for each PeripheralLogicExpression. The relationship between the structures for PeripheralLogicExpression and TransitionLogicExpression are shown in Figure 12: Logic expression containing structures.

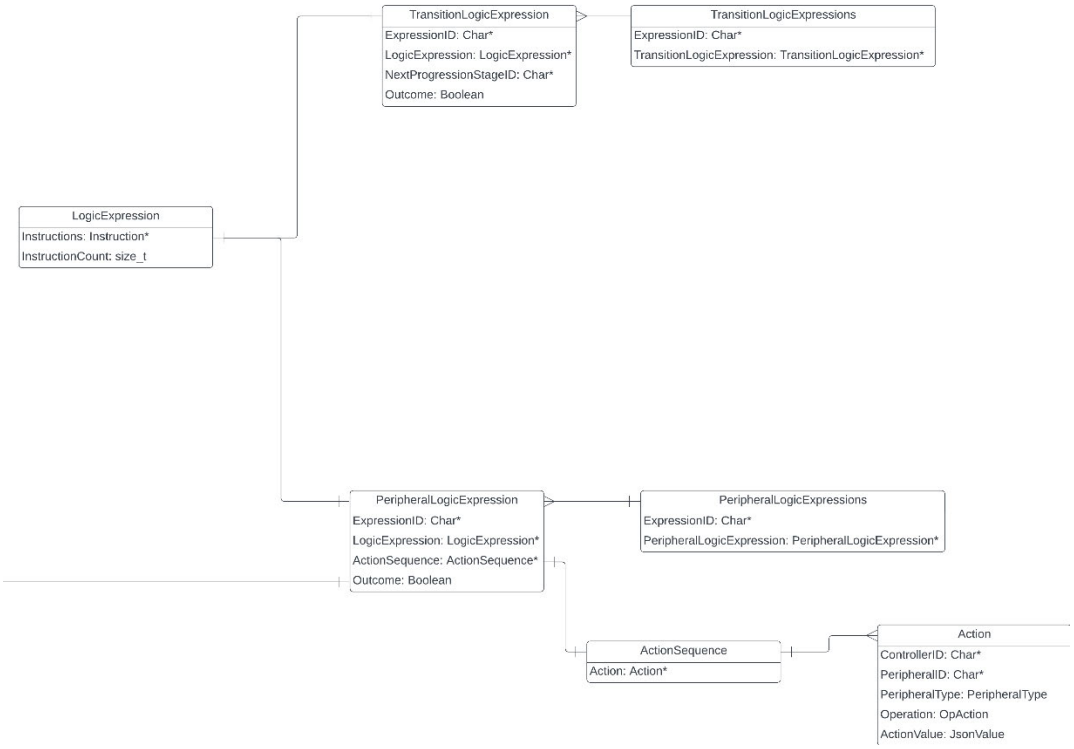


Figure 12: Logic expression containing structures

A representation of a nested JSON expression structure is shown within Source Code Snippet 15: Nested JSON expression representation.

```
// (A AND ((B >= B.ReferenceConstant ) OR ( C < D )) AND E)
{
    // The expression ID of the new expression
    "ExpressionID": "SlidersAndButton",
    "Operator": "AND",
    "Operands": [
      {
        "ControllerID": [String],
        "PeripheralID": [String],
        "ExpressionID": String,
        "PeripheralType": String,
        "OperandDataType": String,
        "ReferenceConstant": [Number, String, Boolean, Array, Object, Null]
      },
      {
        "Operator": "OR",
        "Operands": [
            {
                "Operator": "GTE",
                "Operands": [
                    {
                        "ControllerID": String,
```

```
                        "PeripheralID": String,
                        "ExpressionID": String,
                        "PeripheralType": String,
                        "OperandDataType": String,
                        "ReferenceConstant": [Number, String, Boolean, Array,
Object, Null]
                    }
                ]
            },
            {
                "Operator": "LT",
                "Operands": [
                    {
                        "ControllerID": String,
                        "PeripheralID": String,
                        "ExpressionID": String,
                        "PeripheralType": String,
                        "OperandDataType": String,
                        "ReferenceConstant": [Number, String, Boolean, Array,
Object, Null]
                    },
                    {
                        "ControllerID": String,
                        "PeripheralID": String,
                        "ExpressionID": String,
                        "PeripheralType": String,
                        "OperandDataType": String,
                        "ReferenceConstant": [Number, String, Boolean, Array,
Object, Null]
                    }
                ]
            }
        ]
      },
      {
            "ControllerID": String,
            "PeripheralID": String,
            "ExpressionID": String,
            "PeripheralType": String,
            "OperandDataType": String,
            "ReferenceConstant": [Number, String, Boolean, Array, Object,
Null]
      }
    ]
  }
```

Source Code Snippet 14: Nested JSON expression representation

Table 9: Unit Test UT11 outlines the test cases in which the control logic parser routine must pass in order to meet the objectives and outcomes for the hybrid control system. The test cases are outlined within Table 42: UT11 test case.

Table 42: UT11 test case outcomes

Test Case	Test Description	Outcome
Stage 01	System context structure DeviceProfiles, ExpressionOperands, PeripheralLogicExpressions and TransitionLogicExpressions memory released.	Pass
Stage 02	JSON progression logic stage DeviceProfiles parsed into DeviceProfiles system context structure.	Pass
Stage 03	JSON progression logic stage PeripheralLogicExpressions pared into and ExpressionOperands and system context PeriphralLogicExpressions structure.	Pass
Stage 04	JSON progression logic stage TransitionLogicExpressions pared into system context TransitionLogicExpressions structure.	Pass

The control_logic_interpreter routine is the component within the control logic engine which evaluates the LogicExpressions generated by the control_logic_parser. The control logic interpreter routine contain the evaluate_expresion helper function which is shown in Source Code Snippet 15: evaluate_expression helper function.

```
bool evaluate_expression(const LogicExpression* expression);
```

Source Code Snippet 15: evaluate_expression helper function

The evaluate_expression helper function ingests a target LogicExpression structure and then sequentially executes the instruction OpCodes which either loads operands to the stack or evaluates operands against the instruction OpCode.

Table 10: Unit Test UT12 outlines the test cases in which the control logic interpreter routine must pass in order to meet the objectives and outcome of a control system capable of hybrid progrssion logic. The test cases are outlined within Table 43: UT12 test case outcomes.

Table 43: UT12 test case outcomes

Test Case	Test Description	Outcome
Stage 01	Correct evalution of PeripheralLogicExpression with flat logic with OP_AND, OP_OR.	Pass
Stage 02	Correct evalution of PeripheralLogicExpression with nested logic with OP_AND, OP_OR, OP_NOT and OP_IDENTITY operands.	Pass
Stage 03	Correct evalution of TransitionLogicExpression with flat logic with OP_AND and OP_OR.	Pass
Stage 04	Correct evalution of TransitionLogicExpression with nested logic with OP_AND, OP_OR, OP_NOT and OP_IDENTITY operands.	Pass

Table 12: Unit Test UT14 outlines the test cases in which the master controller's control logic action dispatcher must pass in order to meet the objectives and outcomes. The resulting outcome from each test case in the unit test is shown in Table 44: UT13 outcome matrix.

Table 44: UT13 outcome matrix

Test Case	Test Description	Outcome
Stage 01	ActionSequence containing PeripheralUpdate commands from PeripheralLogicExpression sequentially call communication action dispatcher with Action reference.	Pass
Stage 02	StageComplete Action calls communication action dispatcher with Action reference.	Pass
Stage 03	StageUpdate Action calls communication action dispatcher with Action reference.	Pass

Table 11: Unit Test UT13 outlines the test cases in which the master controller's control logic state manager must pass in order to meet the objectives and outcomes. The resulting outcome from each test case in the unit test is shown in Table 45: UT13 outcome matrix.

Table 45: UT13 outcome matrix

Test Case	Test Description	Outcome
Stage 01	Updates ExpressionOperand instances according to system context.	Pass
Stage 02	Iteratively evaluates each PeripheralLogicExpression within system context structure PeripheralLogicExpressions.	Pass
Stage 03	Iteratively evaluates each TransitionLogicExpression within system context structure TransitionLogicExpressions.	Pass
Stage 04	TransitionLogicExpression evaluation with outcome of true sets next progression logic stage.	Pass
Stage 05	TransitionLogicExpression evaluation with outcome of true sets next progression logic stage.	Pass
Stage 06	TransitionLogicExpression evaluation with outcome of true reads next progression logic stage JSON object from SD card.	Pass
Stage 07	Next progression logic stage updates ProgressionStage system context structure.	Pass
Stage 08	Next progression logic stage JSON object is called by reference to control_logic_parser.	Pass

4.7. Slave Controller's Control Logic Engine

The slave controller's logic expressions are categorised into sequence logic expressions and peripheral logic expressions. Both of these categories have operands which reference the state of pin peripherals or virtual peripherals associated with the current progression logic stage. As some user interactions may require the input be captured sequentially or converted into another data type, the virtual peripherals function as a buffer data structure to store the result of sequential actions. Table 13: Unit Test UT15 outlines the test cases in which the slave controller's control logic parser must pass in order to meet the objectives and outcomes. The resulting outcome from eachs test case in the unit test is shown in Table 46: UT15 outcome matrix.

Table 46: UT15 outcome matrix

Test Case	Test Description	Outcome
Stage 01	System context structure VirtualPeripherals, SequenceLogicExpressions and PeripheralLogicExpressions memory released.	Pass
Stage 02	JSON progression logic stage PeripheralLogicExpressions pared into VirtualPeripherals and the system context structure PeriphralLogicExpressions.	Pass
Stage 03	JSON progression logic stage SequenceLogicExpressions pared into ActionSequence and system context SequenceLogicExpressions structure.	Pass

Table 14: Unit Test UT16 outlines the test cases in which the slave controller's control logic interpreter must pass in order to meet the objectives and outcomes. The resulting outcome from each test case in the unit test is shown in Table 47: UT16 outcome matrix.

Table 47: UT16 outcome matrix

Test Case	Test Description	Outcome
Stage 01	Correct evalution of SequenceLogicExpression with flat logic with OP_AND, OP_OR.	Pass
Stage 02	Correct evalution of SequenceLogicExpression with nested logic with OP_AND, OP_OR, OP_NOT, OP_IDENTITY, OP_EQ, OP_NEQ, OP_GT, OP_GTE, OP_LT, OP_LTE operands.	Pass
Stage 03	Correct evalution of SequenceLogicExpression with flat logic with OP_AND and OP_OR.	Pass
Stage 04	Correct evalution of SequenceLogicExpression with nested logic with OP_AND, OP_OR, OP_NOT, OP_IDENTITY, OP_EQ, OP_NEQ, OP_GT, OP_GTE, OP_LT, OP_LTE operands.	Pass

Table 16: Unit Test UT18 outlines the test cases in which the slave controller's control logic action dispatcher must pass in order to meet the objectives and outcomes. The resulting outcome from each test case in the unit test is shown in Table 48: UT18 outcome matrix.

Table 48: UT18 outcome matrix

Test Case	Test Description	Outcome
Stage 01	ActionSequence containing VPUpdate Actions from SequenceLogicExpressions call iteratively update_virtual_peripheral helper function.	Pass
Stage 02	Action containing OP_WRITE operation overwrites the VirtualPeripheral.PeripheralValue with Action.ActionValue.	Pass
Stage 03	Action containing OP_APPEND operation appends Action.ActionValue to the end of the target VitrualPeripheral.PeripheralValue.	Pass
Stage 04	Action containing OP_CLEAR operation clears the the target VitrualPeripheral.PeripheralValue to default value.	Pass

Table 17: Unit Test UT19 outlines the test cases in which the slave controller's control logic state manager must pass in order to meet the objectives and outcomes. The resulting outcome from each test case in the unit test is shown in Table 49: UT19 outcome matrix.

Table 49: UT19 outcome matrix

Test Case	Test Description	Outcome
Stage 01	Calls peripheral_update routine to enqueue lastest peripheral values.	Pass
Stage 02	Iteratively evaluates each SequenceLogicExpression within system context structure SequenceLogicExpressions.	Pass

Stage 03	Calls control_logic_action_dispatcher routine for each SequenceLogicExpression which evaluates to true.	Pass
Stage 04	Iteratively evaluates each PeripheralLogicExpression within system context structure PeripheralLogicExpressions.	Pass
Stage 05	Set ProgressionStage.UpdateMaster to true if PeripheralLogicExpression outcome evalutes to true.	Pass
Stage 06	Update to progression logic stage ID in ProgressionStage.StageID reads next progression logic stage JSON object from SD card.	Pass
Stage 07	Calls control_logic_parser with progression logic stage JSON object.	Pass
Stage 08	ProgressionStage system context structure assigned ProgressionStage.Configured once progression logic stage is parsed by control_logic_parser.	Pass

4.8. Communication Engine

Table 17: Unit Test UT19 outlines the test cases in which the master controller's communication action dispatcher must pass in order to meet the objectives and outcomes. The resulting outcome from each test case in the unit test is shown in Table 50: UT19 outcome matrix.

Table 50: UT19 outcome matrix

Test Case	Test Description	Outcome
Stage 01	PeripheralUpdate Action from control_logic_action_dispatcher constructs PeripheralUpdate command message in json_schema_serialiser.	Pass
Stage 02	StageComplete Action from control_logic_action_dispatcher constructs	Pass

	StageComplete command message in json_schema_serialiser.	
Stage 03	StageUpdate Action from control_logic_action_dispatcher constructs StageUpdate command message in json_schema_serialiser.	Pass
Stage 04	ConfigStatus Action from configuration_interpreter constructs ConfigStatus command message in json_schema_serialiser.	Pass
Stage 05	Dispatches command message returned from json_schema_serliaser to ble_interface with target ControllerID.	Pass
Stage 06	Prioritises ranking of CommandMessage structure on CommandMessageQueue with ControllerID.	Fail
Stage 07	The priority ranking of CommandMessages on CommandMessageQueue is StageComplete, StageUpdate, PeripheralUpdate.	Fail
Stage 08	Update ExpressionOperands with parsed UpdateOutcome command message.	Pass
Stage 09	Update DeviceProfile.ConfigurationStatus with parsed ConfigStatus command message.	Pass
Stage 10	Store processed CommandMessage to SD card storage CommunicationDump log file.	Pass

Table 18: Unit Test UT20 outlines the test cases in which the slave controller's action dispatcher must pass in order to meet the objectives and outcomes. The resulting outcome from each test case in the unit test is shown in Table 51: UT20 outcome matrix.

Table 51: UT20 outcome matrix

Test Case	Test Description	Outcome
Stage 01	ConfigStatus Action construct ConfigStatus command message in json_schema_serlialiser.	Pass

Stage 02	UpdateOutcome Action from control_logic_action_dispatcher constructs UpdateOutcome command message in json_schema_serlialiser.	Pass
Stage 03	Dispatches command message returned from json_schema_serliaser to ble_interface with target ControllerID.	Pass
Stage 04	Prioritises ranking of CommandMessage structure on CommandMessageQueue with ControllerID.	Fail
Stage 05	The priority ranking of CommandMessages on CommandMessageQueue is StageComplete, StageUpdate, PeripheralUpdate.	Fail
Stage 06	Update PinPeripheral.PeripheralValue for pin peripherals configured with DIRECTION_OUTPUT when UpdatePeripheral command message parsed.	Pass
Stage 07	Update ProgressionStage.StageComplete in system context structure when StageComplete Action parsed.	Pass
Stage 08	Update ProgressionStage.StageID in system context structure when StageUpdate Action parsed.	Pass
Stage 09	Store processed CommandMessage to SD card storage CommunicationDump log file.	Pass

Table 19: Unit Test UT21 outlines the test cases in which the master controllers message parser must pass in order to meet the objectives and outcomes. The resulting outcome from each test case in the unit test is shown in Table 52: UT21 outcome matrix.

Table 52: UT21 outcome matrix

Test Case	Test Description	Outcome
Stage 01	Parse ConfigStatus command message into CommandMessage structure.	Pass
Stage 02	Parse UpdateOutcome command message into CommandMessage structure.	Pass

Table 19: Unit Test UT21 outlines the test cases in which the slave controller's command message parser must pass in order to meet the objectives and outcomes. The resulting outcome from each test case in the unit test is shown in Table 53: UT21.2 outcome matrix.

Table 53: UT21.2 outcome matrix

Test Case	Test Description	Outcome
Stage 01	Parse UpdatePeripheral command message into CommandMessage structure.	Pass
Stage 02	Parse StageComplete command message into CommandMessage structure.	Pass
Stage 03	Parse StageUpdate command message into CommandMessage structure.	Pass
Stage 04	Parse ConfigStatus command message into CommandMessage structure.	Pass

Table 20: Unit Test UT23 outlines the test cases in which the master controller's command message interpreter must pass in order to meet the objectives and outcomes. The resulting outcome from each test case in the unit test is shown in Table 54: UT23 outcome matrix.

Table 54: UT23 outcome matrix

Test Case	Test Description	Outcome
Stage 01	ConfigStatus CommandMessage structure maps to the sub-routine update_config_status in command_action_dispatcher.	Pass
Stage 02	UpdateOutcome CommandMessage structure maps to the sub-routine update_operands in command_action_dispatcher.	Pass

Table 21: Unit Test UT24 outlines the test cases in which the slave controller's command message interpreter must pass in order to meet the objectives and

outcomes. The resulting outcome from each test case in the unit test is shown in Table 55: UT24 outcome matrix.

Table 55: UT24 outcome matrix

Test Case	Test Description	Outcome
Stage 01	UpdatePeripheral CommandMessage structure maps to the sub-routine update_peripheral_output in command_action_dispatcher.	Pass
Stage 02	StageComplete CommandMessage structure maps to the sub-routine update_stage_complete in command_action_dispatcher.	Pass
Stage 03	StageUpdate CommandMessage structure maps to the sub-routine update_stage_id in command_action_dispatcher.	Pass

4.9. Master Controller and Slave Controller Communication Engine

Table 22: Integration Test IT03 outlines the test cases in which the master controller and slave controller's communication engine integration must pass in order to meet the objectives and outcomes. The resulting outcome from each test case in the unit test is shown in Table 56: IT03 outcome matrix.

Table 56: IT03 outcome matrix

Test Case	Test Description	Outcome
Stage 01	Master controller establishes BLE connection with slave controller.	Pass
Stage 02	Master controller sends UpdatePeripheral to slave controller. The slave controller's command_action_dispatcher calls update_peripheral_output helper function.	Pass
Stage 03	Master controller sends StageComplete to slave controller. The slave controller's command_action_dispatcher calls update_stage_complete helper function.	Pass

Stage 04	Master controller sends StageUpdate to slave controller. The slave controller's command_action_dispatcher calls update_stage_id helper function.	Pass
Stage 05	Master controller sends ConfigStatus to slave controller. The slave controlller's command_action_dispatcher calls get_config_status helper function.	Pass
Stage 06	Slave controller sends ConfigStatus to master controller. The master controller's command_action_dispatcher calls get_config_status helper function.	Pass
Stage 07	Slave controller sends UpdateOutcome to master controller. The master controller's command_action_dispatcher calls update_operands helper function.	Pass

4.10. Master Controller Initialisation and Slave Controller Initialisation

Table 23: Integration Test IT04 outlines the test cases in which the master controller's initalisation and slave controller's initalisation integration must pass in order to meet the objectives and outcomes. The resulting outcome from each test case in the unit test is shown in Table 57: IT04 outcome matrix.

Table 57: IT04 outcome matrix

Test Case	Test Description	Outcome
Stage 01	The master_config.json file is parsed into DeviceProfiles system context structure.	Pass
Stage 02	Each slave controller parses the slave_config.json file into DeviceProfiles and PinPeripherals system context structure.	Pass
Stage 03	Master controller establishes connection with three slave controller devices within DeviceProfiles.	Pass
Stage 04	Master controller configuration_interpreter routine sequentially sends ConfigStatus to each slave	Pass

	controller until all DeviceProfile.ConfigurationStatus is true on master controller's DeviceProfiles.	
Stage 05	Slave controller command_action_dispatcher calls get_config_status and sends ConfigStatus command message to master controller.	Pass
Stage 06	Master controllers command_action_dispatcher calls update_config_status helper function with slave controller's configuraiton outcome.	Pass

4.11. Master Controller's Configuration Engine and Control Logic Engine Integration

Table 24: Integration Test IT05 outlines the test cases in which the master controller's configuration engine and control logic engine integration must pass in order to meet the objectives and outcomes. The resulting outcome from each test case in the unit test is shown in Table 58: IT05 outcome matrix.

Table 58: IT05 outcome matrix

Test Case	Test Description	Outcome
Stage 01	The master_config.json file is parsed into DeviceProfiles system context structure.	Pass
Stage 02	Each slave controller parses the slave_config.json file into DeviceProfiles and PinPeripherals system context structure.	Pass
Stage 03	Master controller establishes connection with three slave controller devices within DeviceProfiles.	Pass
Stage 04	Master controller establishes connection with three slave controller devices within DeviceProfiles.	Pass
Stage 05	Master controller configuration_interpreter routine sequentially sends ConfigStatus to each slave controller until all DeviceProfile.ConfigurationStatus is true on master controller's DeviceProfiles.	Pass

Stage 06	Slave controller command_action_dispatcher calls get_config_status and sends ConfigStatus command message to master controller.	Pass
Stage 07	Master controllers command_action_dispatcher calls update_config_status helper function with slave controller's configuraiton outcome.	Pass
Stage 08	Master controller's main routine calls control_logic_state_manager with DeviceProfile.ConfigurationStatus = true and ProgressionStage.StageID = 0.	Pass
Stage 09	Master controller's control_logic_state_manager reads control_logic.json from SD card.	Pass
Stage 10	Mater controller parses and configures progression logic stage with StageID = 0.	Pass

4.12. Master Controller's Control Logic Engine and Communication Engine Integration

Table 25: Integration Test IT06 outlines the test cases in which the master controller's control logic engine and communication engine integration must pass in order to meet the objectives and outcomes. The resulting outcome from each test case in the unit test is shown in Table 59: IT06 outcome matrix.

Table 59: IT06 outcome matrix

Test Case	Test Description	Outcome
Stage 01	Master control logic state manager calls control_logic_parser to parse current progression logic stage from control_logic.json. DeviceProfiles, ExpressionOperands, PeripheralLogicExpressions, TransitionLogicExpression and ProgressionStage system context structures populated with correct objects.	Pass

Stage 02	Master controller's state manager calls command_action_dispatch routine for StageUpdate CommandMessage.	Pass
Stage 03	Master controller's command action dispatcher sequentially sends StageUpdate CommandMessage to each slave controller in DeviceProfiles system context.	Pass
Stage 04	The UpdatePeripheral CommandMessage from each slave controller is received and updates ExpressionOperands system context.	Pass
Stage 05	Master controller's state manager calls control_logic_interpreter with transition logic expression evaluation true.	Pass
Stage 06	Master controller's state manager calls command_action_dispatcher routine for StageComplete.	Pass
Stage 07	Master controller's command_action_dispatcher sequentially sends StageComplete CommandMessage to each slave controller in DeviceProfiles system context.	Pass

4.13. Slave Controller's Control Logic Engine and Communication Engine Integration

Table 26: Integration Test IT07 outlines the test cases in which the slave controller's control logic engine and communication engine integration must pass in order to meet the objectives and outcomes. The resulting outcome from each test case in the unit test is shown in Table 60: IT07 outcome matrix.

Table 60: IT07 outcome matrix

Test Case	Test Description	Outcome
Stage 01	Slave control logic state manager calls control_logic_parser to parse current progression logic stage from control_logic.json. PinPeripherals,	Pass

	PeripheralLogicExpressions, SequenceLogicExpressions and ProgressionStage system context structures populated with correct objects.	
Stage 02	Master controller's state manager calls peripheral_update routine and begin evaluation cycle.	Pass
Stage 03	When ProgressionStage.MasterUpdate set to true slave controller's state manager calls command_action_dispatcher for UpdateOutcome CommandMessage.	Pass
Stage 04	The command_action_dispatcher routine returns to control_logic_state_manager and evaluation cycle continues.	Pass
Stage 05	The control logic enginer parses StageComplete CommandMessage and calls update_stage_complete in command_action_dispatcher. The main state changes to idle.	Pass
Stage 06	The control logic enginer parses StageUpdate CommandMessage and calls update_stage_id in command_action_dispatcher. The ProgressionStage.StageID is set to new progression logic stage ID and calls control_logic_state_manager routine.	Pass

4.14. Operational Escape Room Purpose Testing

Figure 13: ST01 hybrid progression path shows the progression logic path developed for the control_logic.json files in order to conduct ST01.

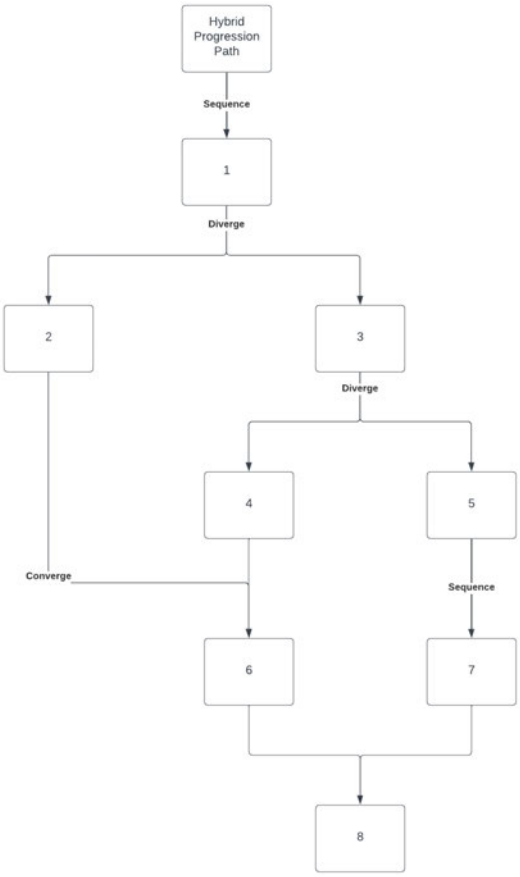


Figure 13: ST01 hybrid progression path

Table 27: System Test ST01 outlines the test cases in which the operational escape room purpose testing must pass in order to meet the objectives and outcomes. The resulting outcome from each test case in the unit test is shown in Table 61: ST01 outcome matrix.

Table 61: ST01 outcome matrix

Test Case	Test Description	Outcome
Stage 01	Master controller configuration engine initalises system context structures and reads master_config.json from SD card.	Pass
Stage 02	Master controller configuration engine parses DeviceProfiles and configuration_interpreter routine.	Pass

Stage 03	Master controller establishes connection with the three slave controller devices within DeviceProfiles.	Pass
Stage 04	The three slave controllers call their configuration engine and initalises system context structures from SD card slave_config.json.	Pass
Stage 05	Each slave controller's configuration engine parses PinPeripherals and updates DeviceProfile.ConfigurationStatus.	Pass
Stage 06	Master control logic engine configures progression logic stage ID = 0.	Pass
Stage 07	Master controller sends StageUpdate to each slave controller in DeviceProfiles for current progression logic stage.	Pass
Stage 08	Slave controllers relevant to stage call their control logic engine to configure progression logic stage ID = 0.	Pass
Stage 09	User interactions with slave controller captured by periphreal_update and associated PinPeripheral.PeripheralValue updated.	Pass
Stage 10	Slave controller's PeripheralLogicExpression outcome updates trigger MasterUpdate.	Pass
Stage 11	Master controller receiving UpdateOutcome updates ExpressionOperands.	Pass
Stage 12	Slave controller's receiving UpdatePeripheral CommandMessage changes output peripherals PinPeripheral.PeripheralValue state. Can be seen visibly at LED output.	Pass
Stage 13	Correct evaluation of TransitionLogicExpression updates system context	Pass
Stage 14	Progression logic stage transitions occur according to the hybrid progression path in TransitionLogicExprssions.	Pass
Stage 15	Progression StageID path validated: 1->2->6->8	Pass

Stage 16	Progression StageID path validated: 1->3->4->6->8	Pass
Stage 17	Progression StageID path validated: 1->3->5->7->8	Pass

CHAPTER 5: DISCUSSION

5.1. Introduction to Discussion

5.1.1. Research Objectives and Outcomes

Evaluating the project's objectives and outcomes against the results of the methodology's development process enables the suitability of the control system to be determined for the escape room industry. The project achieved all objectives and outcomes by integrating the necessary components into a cohesive system design. Each component within the control system contributes towards achieving multiple objectives and outcomes. The discussion relates the project outcomes and objectives to the key findings from relevant results.

Specific Objectives:

- The design of a low-cost master-slave control system architecture is suitable for escape room puzzles.
- Create a flexible escape room, progression logic file type and data structure that can be loaded onto the master and slave controllers via SD card.
- Implement master and slave interpreter for the game progression logic file data structure, ensuring accurate game progression and puzzle state management.
- Investigate coordination and scheduling schemes for master-slave communication, prioritising puzzle querying based on game progression.
- Analyse how different data file schemes impact system scalability and complexity.
- Evaluate and implement appropriate wireless communication methods, protocols and technologies for the master and slave controllers.
- Investigate system usability limitations. Analysing system latency, data rate, loss tolerance, wireless communication distance and response time.
- Evaluation of communication strategies such as suitability of polling versus interrupt-based methods for communication between master and slave devices.

  If time permits,

- Develop a graphical web application to create the game progression logic.
- Develop a compiler that compiles the game progression logic into the master and slave game progression logic file.

Expected Outcomes:

- Low-cost, master controller embedded system with wireless communication, SD card reader and game file interpreter.
- Low-cost, slave controller embedded system capable of wireless communication, SD card reader, game file interpreter and peripheral API for puzzle control.
- A robust communication method and data protocol tailored for master-slave interactions within the escape room environment.
- Definition and implementation of a universal game progression logic file data structure and file type that supports complex escape room game design.
- A game progression logic file interpreter on both master and slave devices, ensuring correct puzzle state management, game progression and data transfer.
- Implementation of an effective coordination scheme in the master controller to manage and query slave devices based on the current stage of the game progression.
- Understand the usage limitations and scalability of the modular control system.

  If time permits,
- A web application for designing escape room game logic.
- Error handling and input validation of web application design tool user input.
- Implementation of a compiler to generate required game file format for master and slave controllers from graphical web application representation.

5.1.2. Overview of Key Findings

The control system analysis evaluated the developed components' response in isolation, and integration, within an operational escape room environment and under performance testing conditions. The validation of the control systems components at each level confirms that the system can reliably meet the requirements of the escape

room industry. The development approach allows the separation of concerns within the developed source code. The separation of concern is achieved by developing each component as modular and isolated routines, where each component does not impact another through run-time side effects. This provides reliable log file analysis, source code maintenance and debugging.

5.2. Interpretation of Results

The control logic engine and configuration engine could successfully parse the JSON configuration files into the internal data structures for flexible escape room narrative use cases. The master_config.json and slave_config.json files were parsed and interpreted by the configuration parser and defined the device profile or peripheral profile for the loaded narrative. The configuration engine ensured that all slave controllers within the narrative had been successfully configured before the first escape room progression stage was loaded. The results from the control system configuration prove to be suitable for escape room usage as the master controller's state manager evaluates the narrative progression through four categories of logic expression. The master controller's peripheral logic expression evaluates the system-wide peripheral state of all slave controllers within the current progression logic stage. This allows the one controller to meet the many puzzle requirements discovered during the literature review. The peripheral logic expressions also enable puzzle peripheral updates to be actioned without impacting the progression of the narrative. This allows for the engaging experience required by the escape room narratives through updating the environment state. The transition logic expressions evaluate the outcome of its peripheral logic expressions. The transition logic expressions successfully evaluated the peripheral logic expressions' outcomes, allowing for complex system-wide logic expressions to determine hybrid narrative progression. The hybrid or flexible progression logic outcome was achieved by successfully testing the peripheral logic expressions and transition logic expressions within the configuration engine and control logic engine test cases. Time did not permit the communication coverage and latency results for the Bluetooth BLE performance tests. Despite this, the Bluetooth BLE was proven to be an effective wireless technology for the escape room use case, as shown in Table 37: UT04 Outcome Matrix. Bluetooth BLE can manage multiple slave controller connections simultaneously when the master controller is configured as a central device and the slave controllers as a peripheral.

5.3. Implications of Findings

5.3.1. Implications for educational and recreational accessibility.

The findings from the project fulfil the escape room industry's specific need for a scalable, low-cost control system that does not require embedded programming and electronic skills to create complex narratives. The study by Sánchez-Martín et al. (2020) suggested that the ease of implementing complex narratives could broaden the adoption of immersive learning in educational settings. The successful system tests within the project demonstrate the control system's ability to configure complex narrative progression into the escape room without programming embedded hardware. Increasing peripheral and narrative logic expression complexity did not increase the setup complexity or limit the control system's ability to scale. By simplifying the setup and configuration, the control system will make escape room activities more accessible to educators by reducing the technical and financial barriers. This highlights the benefit of enabling a more engaging educational experience that promotes collaborative problem-solving for low-motivation topics (Gordon et al., 2019).

5.3.2. Implications for control system scale and flexibility in escape rooms.

Master-Slave Architecture Multiplicity

The control system's master-slave architecture supports one-to-many relationships, enabling a single master to manage multiple slave controllers effectively. As Ross & Bennett (2022) outlined, an escape room narrative can contain many puzzles requiring that the control system's performance does not degrade when managing multiple puzzle states. Ross (2019) demonstrated that each puzzle within an escape room control system can have multiple input and output peripherals. This requires that the control system suitable for escape room usage must be able to manage the narrative progression with multiple puzzles containing multiple input and output peripherals. The control system's master controller was shown to manage multiple slave controller connections concurrently. This demonstrates the control system's ability to manage multiple puzzles' peripheral input and output states distributed throughout an escape room facility. This was due to the master-slave architecture allowing each slave controller to manage its local puzzle peripherals independently of the master controller. The combined system context evaluation of multiple slave controllers is then

evaluated within the central master controller's control logic engine. This design decision delegates the evaluation of peripheral logic expressions local to the connected slave controller. This highlights that the control system's architecture directly contributes to achieving the scalability requirements of an escape room use case.

5.4. Limitations of Study

5.4.1. Overview of Limitations

The study's limitations impact the broader applicability of the project's outcomes and findings. While the system met the defined functional objectives and outcomes, performance may differ within actual escape room facilities when considering system communication characteristics, reliability and power stability. This is due to the test cases being conducted within controlled operational conditions. Despite these limitations, the study provides a solid foundation for further research and refinement, especially in operational settings and user-driven scenarios.

5.4.2. Industry Specific Limitations

The research design targets successfully implementing and testing control system components for escape room usage. Despite this, multiple design decisions impacted the study's understanding of various system performances and suitability in broader operational conditions. The scope of the study's test cases was conducted within controlled conditions instead of real-world escape room facilities. This limits the study's ability to understand factors like fluctuating user interactions and environmental influences that may be specific to escape room facilities. The operational system tests were designed to understand the control system's response to untrained user interactions. However, factors relating to the physical environment and design of escape room narratives aren't considered within the test stages. Specifically, some escape room facilities decorate the controlled environment to promote an engaging experience. This could result in the slave controller being contained within decorative materials, impacting wireless communication characteristics. The distance between slave controllers could also be significant within different escape room narratives. This could potentially have an impact when multiple slave controllers are distributed uneven distances throughout the room and are evaluated within the same master controller

logic expression. The research design doesn’t allow for such use cases to be analysed.

5.4.3. Limitations of Test Regime

The test regime's system and performance test stages were crucial for evaluating the isolated operational characteristics of the control system. Isolating the system and performance test cases limits understanding of how performance metrics like communication latency, coverage, and memory usage would behave under operational, real-world escape room usage conditions. Combining the performance metric test cases with operational system testing and user interactions would yield a more comprehensive insight into realistic escape room scenarios.

5.4.4. Limitations of Design Decisions

The scope methodology’s development process also constrains the performance and test case results to the specific source code design decisions. The results don't consider the impact of programming decisions such as selected data structures, employed algorithms, CPU scheduling routines, and resource allocation. The design decisions around resource utilisation, algorithms and data structures can impact the time complexity of each developed routine. An example of this within the project is using table mapping data structures to search for specific data structures based on key pairs. An alternative approach could be to use hash tables to achieve O(1) search time complexity for desired data structure values. Another programming design decision was to use recursive algorithms instead of iterative for nested JSON traversal. Due to project time constraints, the study does not consider the impact of different programming approaches. Therefore, simplicity was selected instead of efficiency for some implementation decisions. Another implementation limitation of the study is that robust logic error checking is not employed to check for memory leaks during component integration. Therefore, orphaned memory or memory leaks have the potential to impact run-time performance and operation. To alleviate this limitation, each routine frees memory resources managed locally before returning to the calling function. This design decision ensures that memory leaks are contained within the running routine's scope.

5.4.5. Limitations of Constrained Scope

While the power management and interface electronics were carefully designed for the controlled use cases within the project, power stability and load conditions were not considered to understand their impact on system operation. This limits the insight into system performance when expanding peripheral scale and power requirements. The scope of the research design context focused on configuring the peripheral state and values. However, the embedded microcontroller becomes unreliable as peripheral power requirements change outside of the scope of the controlled case studies.

5.5. Suggestions for Future Research

The development of a graphical web application to create and compile the game progression logic is a concept proposed within the objectives and outcomes of this project if time permits. Developing a graphical web application interface would simplify generating the configuration files, allowing non-technical users to design the entire escape room narrative quickly and easily. Successful implementation of this project outcome would allow for increased system usability and adoption while reducing configuration time. The potential industry impact of this change would result in a standardised game configuration across all escape rooms, promoting broader adoption of the configurable control system. The suggested approach is to develop a drag-and-drop interface, which directly translates the graphical programming language into the JSON configuration files for the master and slave controllers. Another suggestion for future research is a proposed approach for run-time monitoring of the control system. The proposed control system design requires each controller's SD card to be ejected to monitor or view log files. This results in the escape room's inability to monitor the control system’s log file output during run-time usage. The suggested approach to resolve this would be to investigate a host device that can receive the command message broadcast over Bluetooth BLE. The final suggestion for future research is further research into optimised power solutions and interface electronics for the peripheral microcontroller of slave controllers. The project proposes an embedded system capable of configuring the pin peripherals of the peripheral microcontroller; however, the current implementation is sensitive to very low voltage and current conditions. Investigating a more robust power regulation and interface circuitry would reduce the complexity and associated risk with power electronics for non-technical escape room designers.

CONCLUSION

The project presented the design, development, testing and evaluation of a scalable, low-cost master-slave control system for the escape room industry. The critical objectives sought to implement a flexible progression logic system, ensure robust wireless communication and address the technical limitations that make the escape room less accessible to non-technical designers. The research demonstrated that the control system achieved all objectives and outcomes for the escape room usage. The modular embedded architecture configured with JSON files enabled automated communication profiles and scalable hybrid narrative progression. Using Bluetooth BLE for wireless communication proved effective for low-cost, multi-room setups. The nested JSON representation of control logic addresses the industry's gap in accessibility for non-technical escape room designers. The study contributes to the escape room industry by presenting a low-cost solution to the resourcing challenges identified in the literature. The configurable control system reduces the technical barriers to implementing complex escape room narratives. The project advances the field by integrating nested JSON configuration files into embedded master-slave architecture for configuration and communication. While the configurable control system met its objectives, limitations were introduced by relying on controlled testing environments that did not reflect real-world escape room facilities. The design decisions within the development process also introduced limitations by trading implementation efficiency for simplicity to meet project timeframes. These limitations suggest areas where the system could be refined to better address operational conditions and optimisation. Future work should focus on system stability and provide further abstraction to generate configuration files. This includes the development of a graphical web application that compiles the configuration files, real-time control system monitoring and robust embedded electronic design. This research provides a strong foundation for implementing a low-cost control system for pop-up escape rooms. Addressing the key challenges demonstrates an innovative approach that offers a solution to make escape room experiences more immersive and accessible.

REFERENCES

Anani, W., Ouda, A., & Hamou, A. (2019). A survey of wireless communications for IOT echo-systems. *IEEE*, 1-6. https://doi.org/10.1109/CCECE.2019.8861764

Angelov, K.K., Kogias, P.G., & Pasarelski, R.I. (2023). Application and performance analysis of lora end devices for monitoring of indoor lighting systems. *IEEE,* 1-5. https://doi.org/10.1109/ET59121.2023.10279194

Bashi, A. (2024). Control system with fuzzy Logic. ResearchGate. https://doi.org/10.5281/zenodo.10640837

Bennetts, R.G. (1982). Analysis of reliability block diagrams by boolean techniques. *IEEE, R-31*(2),159–166. https://doi.org/10.1109/TR.1982.5221283

Cain, J. (2019). Exploratory implementation of a blended format escape room in a large enrollment pharmacy management class, *Currents in Pharmacy Teaching and Learning*, *11*(1), 44–50. https://doi.org/10.1016/j.cptl.2018.09.010

Chang, H.-Y.H. (2019). *Escaping the Gap: Escape Rooms as an Environmental Education Tool*. [online] Berkeley Rausser. https://nature.berkeley.edu/classes/es196/projects/2019final/ChangH_2019.pdf

Chrysalidis, P., & Frank, T. (2024). A universal configuration format for avionics. *AvioSE'24*, 45–54. http://dx.doi.org/10.18420/sw2024-ws_04

Darejeh, A. (2023). Empowering education through EERP: A customizable educational VR escape room platform. *IEEE*, 764-766. https://doi.org/10.1109/ismar-adjunct60411.2023.00166

ELMET Project. (2021). *Guide for design, implementation facilitation and evaluation of educational escape rooms.* Retrieved March 30, 2024. https://www.elmetproject.eu/assets/files/ELMET_Guide_EN.pdf

Elsner, C., Lohmann, D., & Schroder-Preikschat, W. (2011). Fixing configuration inconsistencies across file type boundaries. *IEEE,* 116-123. http://dx.doi.org/10.1109/SEAA.2011.26

Gordon, S.K.,Trovinger, S., & DeLellis, T. (2019). Escape from the usual: Development and implementation of an 'escape room' activity to assess team dynamics. *Currents in Pharmacy Teaching and Learning*, *11*(8),818–824. https://doi.org/10.1016/j.cptl.2019.04.013

Gutjahr, B., & Heumesser, R. (2014). Generating configuration files. European Patent Office. https://patents.google.com/patent/EP1953652A1

Hanou, I., Smitskamp, G., & de Schipper, M. (2020). *Designing an escape room sensory system: S.C.I.L.E.R.: Sensory communication inside live escape rooms. TU* Delft. https://repository.tudelft.nl/record/uuid:31969411-1053-4bc1-830d-

Hacer Tercanli, Martina, R., Amorim, M., Wakkee, I., Reuter, J., Cohen, Y., Cohen, Y., Madaleno, M., Vieira, S., Cláudia Miranda Veloso, Figueiredo, C. P., Andreia Vitória, Isabel Cristina Gomes, Meireles, G., Audrone Daubariene, Asta Daunoriene, Andreas Korntved Mortensen, Zinovyeva, A., Irene Rivera Trigueros, & Abigail López Alcarria. (2021). Educational escape rooms in practice: Research, experiences, and recommendations. *UA Editora.* https://doi.org/10.34624/rpxk-hc61

He, S., Huang, L., Gao, G., Wang, G., Wang, Z., & Chen, X. (2019). Design of real-time control in poloidal field power supply based on finite-state machine. *IEEE*, *47*(4), 1878–1883. https://doi.org/10.1109/TPS.2019.2904796

Ivanescu, N.A., Borangiu, T., Brotac, S., & Dogar, A. (2007). Implementation of sequential function charts with microcontrollers. *IEEE,* 1-6. https://doi.org/10.1109/MED.2007.4433852

Kashiwagi, Y., Harada, H., Masaki, H., & Osumi, K. (2022). Development of evaluation systems for large-scale wi-sun fan-based IOT Applications. *IEEE*, 1-6. https://doi.org/10.1109/PIMRC54779.2022.9977832

Kasbe, T. (2015). Importance & Effective Use of Configuration Filein Software Development (Window /Web) in .NET. *SSRN Electronic Journal*, *2*(11), 90–94. https://www.researchgate.net/publication/354191863_Importance_Effective_Use_of_Configuration_File_in_Software_Development_Window_Web_in_NET

Kinio, A.E., Dufresne, L., Brandys, T., & Jetty, P. (2019). Break out of the classroom: The use of escape rooms as an alternative teaching strategy in surgical education. *Journal of Surgical Education*, *76*(1),134–139. https://doi.org/10.1016/j.jsurg.2018.06.030

Kiruthika, V., Jagadeeswari, M., Prabha, C.S., Vaishnavi, V., & Sreejaa, H. (2022). virtual reality based escape room. *IEEE,* 1-5. https://doi.org/10.1109/ICACTA54488.2022.9753531

Lakomkina, I. (2023). *The impact of technology on escape rooms*. Escape Reality. https://www.escapereality.com/blog/the-impact-of-technology-on-escape-rooms/

Lee, K.S.,, Blair, R.A.,, Dhayagude, N.,, Van de Steeg, K., & Schaffner, H. (2014). *Method and apparatus for distributing configuration files in a distributed control system*. https://patentimages.storage.googleapis.com/fd/23/b2/0c68bd5820befd/US20090172223A1.pdf

Li, P.-Y., Chou, Y.-K., Chen, Y.-J., & Chiu, R.-S. (2018). Problem-based learning (PBL) in interactive design: A case study of escape the room puzzle design. *IEEE*, 250-253. http://dx.doi.org/10.1109/ICKII.2018.8569131

Mallaband, S. (1991). Specification of real time control systems by means of sequential function charts. *IET,* 57-62. https://ieeexplore.ieee.org/document/140047

Miroshnyk, M., Poroshyn, S., Shkil, A., Kulak, E., Filippenko, I., Kucherenko, D., Pakhomov, Y., Juliia, S., & Goga, M. (2018). Design of logical control units based on finite state machines' patterns. *IEEE,* 1-6. https://doi.org/10.1109/EWDTS.2018.8524869

Miroshnyk, M., Shkil, O., Kulak, E., Rakhlis, D., Filippenko, I., Hoha, M., Malakhov, M., & Sergienko, V. (2019). Design of real-time system logic control on FPGA. *IEEE,* 1-4. https://doi.org/10.1109/EWDTS.2019.8884387

Park, J.-I., & Umirov, U. (2012). Efficient use of Bluetooth in networked control systems. *IEEE*, 13-17. https://ieeexplore.ieee.org/document/6393395

Peng, Z., Ma, L., & Xia, F. (2008). A low-cost embedded controller for complex control systems. *IEEE, 23-29*. https://doi.org/10.1109/EUC.2008.40

Remelhe, M.P., Lohmann, S., Stursberg, O., Engell, S., & Bauer, N. (2005). Algorithmic verification of logic controllers given as sequential function charts. *IEEE*, 55-58. https://doi.org/10.1109/CACSD.2004.1393850

Ross, R. (2019). Design of an open-source decoder for educational escape rooms. *IEEE, 7*, 145777–145783. https://doi.org/10.1109/ACCESS.2019.2945289

Ross, R., & Bennett, A. (2022). Increasing engagement with engineering escape rooms. *IEEE Transactions on Games*, *14*(2), 161–169. https://doi.org/10.1109/TG.2020.3025003

Sánchez-Martín, J., Corrales-Serrano, M., Luque-Sendra, A., Zamora-Polo, F. (2020). Exit for success. Gamifying science and technology for university students using escape-room. A preliminary approach. *Heliyon*, *6*(7). https://doi.org/10.1016/j.heliyon.2020.e04340

Shaik, M.I. (2011). Design & implementation of ARM based data acquisition system. *IEEE*, 38-42. https://doi.org/10.1109/ICECCT.2011.6077066

Spira, L. (2023). *US Escape Room Industry Report – December 2023*. *Room Escape Artist*. https://roomescapeartist.com/2023/12/28/us-escape-room-industry-report-december-2023/

Staneva, A., Ivanova, T., Rasheva-Yordanova, K., & Borissova, D. (2023). Gamification in education: Building an escape room using VR Technologies. *IEEE,* 678-683. http://dx.doi.org/10.23919/MIPRO57284.2023.10159923

TARAMAA, J., SEPPÄNEN, V., and MÄKÄRÄINEN, M. (1996). From software configuration to application management—improving the maturity of the maintenance of embedded software. *Journal of Software Maintenance: Research and Practice*, *8*(1), 49–75. https://doi.org/10.1002/(SICI)1096-908X(199601)8:1<49::AID-SMR120>3.0.CO;2-Z

Vojir, M., & Beran, L. (2015). Global data structure for positioning machine controlled by PLC. *IEEE,* 1-6. https://doi.org/10.1109/ECMSM.2015.7208699

Wareham, R. (1988). Ladder diagram and sequential function chart languages in programmable controllers. *IEEE,* 12A-14/1. https://doi.org/10.1109/PROCCE.1988.82231

Wehner, P., Piberger, C., & Gohringer, D. (2014). Using JSON to manage communication between services in the internet of things. *IEEE,* 1-4. https://doi.org/10.1109/ReCoSoC.2014.6861361

Yulin, D., & Chunjiao, Z. (2011). Design and research of embedded PLC development system. *IEEE*, 226-228. https://doi.org/10.1109/ICCRD.2011.5764286

APPENDICES

6.1. Appendix A: Risk Assessment

5572	RISK DESCRIPTION	STATUS	TREND	CURRENT	RESIDUAL
	Low-Cost Control System for Pop-Up Escape Room	Live		Low	Low

RISK OWNER	RISK IDENTIFIED ON	LAST REVIEWED ON	NEXT SCHEDULED REVIEW
Kieran Davey	25/05/2024	19/09/2024	19/09/2025

RISK FACTOR(S)	EXISTING CONTROL(S)	CURRENT	PROPOSED CONTROL(S)	TREATMENT OWNER	DUE DATE	RESIDUAL
Physical fatigue and strain resulting from working at computer for extended periods. -> Eye strain / fatigue. -> Work posture fatigue.	Control: The physical fatigue and strain will be eliminated with scheduling regular breaks into computer work periods. Taking a break every 30 minutes to walk away from the computer desk. During computer breaks, no screen time on other devices are permitted. A standing work station will be utilised to eliminate the risk of work posture fatigue.	Low				Low
Using soldering iron on printed circuit board can result in burning skin. The solder also produces fumes.	Control: Wear long sleeve shirt, soldering gloves, face mask, ducted ventilation and work within well ventilated area. This will eliminate the bare skin which can come into contact with soldering iron.	Low	The ducted fan ventilation ingests the fumes of the solder away from user. Lead free solder will be used to reduce exposure to toxic fumes. The use of surface mount components will be utilised where possible to eliminate exposure to hand held soldering activities.		25/08/2024	Very Low
Fire risk from heated element within soldering iron and electronics circuit boards.	Control: Ensure successful test of smoke alarms. Keep fire extinguisher near work station.	Low			25/08/2024	Low
Interaction with custom extra low voltage electronic circuits which contain components rated between 3.3V - 5V.	Control: Utilise insulated tools rated for electrical use. Wear electrical resistant boots.	Low	Electronic circuits will be contained within 3D printed housing when power is connected. 3D printed housing will create isolation between user and electronic components.		25/08/2024	Very Low
Static electricity damage to electronics.	Control: Anti-static mats and wrist straps to be used when interacting directly with electronic components.	Low			18/09/2024	Low
Showcase of project during	Control: Alleviate confusion by	Low	Remove components which aren't		13/10/2024	Very Low

system testing and presentation may have users confused as to how to interact with escape room. This would be due to them not being locked within an enclosed area. Hazard for unintended use of system and unsafe practises.	providing instructional material at project showcase. Also explain escape room progression indicators before allows users to interact with control system.		required for completion of task from user's field of view. Clear the surrounding area of cables and obstructive surfaces.			

6.2. Appendix B: Budget and List of Materials

Resource Description	Category	Quantity	Source / Supplier	Total Cost ($AUD)
Computer Desktop	Equipment	1	Student	N/A
ESP32-C6-DevKitC-1-N8	Equipment	9	Student	135
SD Card Reader / Writer	Equipment	1	Student	N/A
USB Type B to USB Type A	Equipment	1	Student	N/A
32GB SD Card	Equipment	4	Student	16
SD Card Port Slot	Equipment	4	Student	16
ATMEGA328P	Equipment	8	Student	40
Arduino Uno	Equipment	1	Student	N/A
Number Pad	Equipment	1	Student	4
Linear Potentiometer	Equipment	3	Student	72
Ultrasonic Sensor	Equipment	1	Student	7
5V Voltage Regulator	Equipment	8	Student	15
3D Printer	Equipment	1	Student	N/A
Light Emitting Diode (LED)	Equipment	43	Student	20
Passive / Active Electronic Components	Equipment	N/A	Student	10
Printed Circuit Board Manufactured	Equipment	4	Student	30
Endnote	Software	1	Student	N/A
LaTeX	Software	1	Student	N/A
MATLAB	Software	1	USQ	N/A
ESP-IDF v5.3.1	Software	1	Student	N/A
Altium Designer	Software	1	USQ	N/A
Visual Studio Code	Software	1	Student	N/A
IEEE Xplore	Access	1	USQ	N/A
ScienceDirect	Access	1	USQ	N/A
Springer	Access	1	USQ	N/A
Total Resourcing Cost				365

6.3. Appendix C: Unit Tests

6.3.1. UT01 - Embedded Architecture of Master Controller

Source Code Implementation

File Name: sd_card_interface.h

File Content:

```
#ifndef SD_CARD_INTERFACE_H
#define SD_CARD_INTERFACE_H

// Include esp_err so other routines can recieve sd card error type in return.
#include "esp_err.h"

// Initialise and mount the SD card filesystem.
esp_err_t sd_card_init(void);
// Write string data to path.
esp_err_t sd_card_write_file(const char *path, const char *data);
// Append string data to path.
esp_err_t sd_card_append_file(const char *path, const char *data);
// Read file from path. Dynamic allocation in memory for dynamic file sizes.
esp_err_t sd_card_read_file_dynamic(const char *path, char **buffer, size_t
*file_size);
// Unmount the SD card filesystem and deinitialise
void sd_card_deinit(void);

#endif // SD_CARD_INTERFACE_H
```

File Name: sd_card_interface.c

File Content:

```
// ////////////////////
// Include libraries
// External Components
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/unistd.h>
#include <sys/stat.h>
#include <errno.h>
#include "esp_vfs_fat.h"
#include "sdmmc_cmd.h"
#include "driver/sdspi_host.h"
#include "driver/spi_common.h"
// Custom Components
#include "sd_card_interface.h"

// ///////////
// Definitions
```

```
#define MOUNT_POINT "/sdcard"
// GPIO Assignment for SD Card Interface ESP32-C6-DevKit
#define PIN_NUM_MISO 6   // GPIO6
#define PIN_NUM_MOSI 4   // GPIO4
#define PIN_NUM_CLK  5   // GPIO5
#define PIN_NUM_CS   7   // GPIO7

static sdmmc_card_t *card;
static sdmmc_host_t host = SDSPI_HOST_DEFAULT();

// //////////////////
// Initalise sd card.
esp_err_t sd_card_init(void)
{
    esp_err_t ret;

    // Configure mount.
    esp_vfs_fat_sdmmc_mount_config_t mount_config = {
        .format_if_mount_failed = false,
        .max_files = 5,
        .allocation_unit_size = 16 * 1024,
    };
    // Update host configuration.
    host.slot = SPI2_HOST;
    host.max_freq_khz = SDMMC_FREQ_DEFAULT;
    // Configure SPI bus.
    spi_bus_config_t bus_cfg = {
        .mosi_io_num = PIN_NUM_MOSI,
        .miso_io_num = PIN_NUM_MISO,
        .sclk_io_num = PIN_NUM_CLK,
        .quadwp_io_num = -1,
        .quadhd_io_num = -1,
        .max_transfer_sz = 4000,
    };
    // Initialize SPI bus.
    ret = spi_bus_initialize(host.slot, &bus_cfg, SDSPI_DEFAULT_DMA);
    if (ret != ESP_OK) {
        return ret;
    }
    // Configure SD card slot.
    sdspi_device_config_t slot_config = SDSPI_DEVICE_CONFIG_DEFAULT();
    slot_config.gpio_cs = PIN_NUM_CS;
    slot_config.host_id = host.slot;
    // Mount the filesystem.
    ret = esp_vfs_fat_sdspi_mount(MOUNT_POINT, &host, &slot_config,
&mount_config, &card);
    if (ret != ESP_OK) {
        spi_bus_free(host.slot);
```

```
        return ret;
    }

    return ESP_OK;
}

// ////////////////
// Write to sd card.
esp_err_t sd_card_write_file(const char *path, const char *data)
{
    char full_path[128];
    snprintf(full_path, sizeof(full_path), "%s/%s", MOUNT_POINT, path);

    // Open file for writing.
    FILE *f = fopen(full_path, "w");
    if (f == NULL) {
        return ESP_FAIL;
    }
    // Write data to file.
    size_t data_len = strlen(data);
    size_t bytes_written = fwrite(data, 1, data_len, f);
    fclose(f);
    // Ensure all data was written
    if (bytes_written != data_len) {
        return ESP_FAIL;
    }

    return ESP_OK;
}

// ////////////////
// Append to sd card.
esp_err_t sd_card_append_file(const char *path, const char *data)
{
    char full_path[128];
    snprintf(full_path, sizeof(full_path), "%s/%s", MOUNT_POINT, path);

    // Open file for appending.
    FILE *f = fopen(full_path, "a");
    if (f == NULL) {
        return ESP_FAIL;
    }
    // Append data to file.
    size_t data_len = strlen(data);
    size_t bytes_written = fwrite(data, 1, data_len, f);
    fclose(f);
    // Ensure all data was written
    if (bytes_written != data_len) {
```

```
        return ESP_FAIL;
    }

    return ESP_OK;
}

// ///////////////////////
// Read dynamic size from sd card.
esp_err_t sd_card_read_file_dynamic(const char *path, char **buffer, size_t
*file_size)
{
    char full_path[128];
    snprintf(full_path, sizeof(full_path), "%s/%s", MOUNT_POINT, path);
    // Open file for reading.
    FILE *f = fopen(full_path, "r");
    if (f == NULL) {
        return ESP_FAIL;
    }
    // Determine file size.
    if (fseek(f, 0, SEEK_END) != 0) {
        fclose(f);
        return ESP_FAIL;
    }
    long size = ftell(f);
    if (size < 0) {
        fclose(f);
        return ESP_FAIL;
    }
    *file_size = (size_t)size;
    rewind(f);
    // Allocate read buffer.
    *buffer = malloc(*file_size + 1);
    if (*buffer == NULL) {
        fclose(f);
        return ESP_ERR_NO_MEM;
    }
    // Read file content.
    size_t bytes_read = fread(*buffer, 1, *file_size, f);
    fclose(f);
    // Ensure entire file was read.
    if (bytes_read != *file_size) {
        free(*buffer);
        return ESP_FAIL;
    }
    (*buffer)[bytes_read] = '\0';

    return ESP_OK;
}
```

```
// //////////////////////
// Sd card deinitalise
void sd_card_deinit(void)
{
    esp_vfs_fat_sdcard_unmount(MOUNT_POINT, card);
    spi_bus_free(host.slot);
}
```

Unit Test Source Code Files

File Name: main.c

File Content:

```
// ////////////////////
// Include libraries
// External Components
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// Custom Components
#include "sd_card_interface.h"

int app_main(void)
{
    // Init error type from sdmmc.
    esp_err_t ret;

    // ////////////////////
    // Initialize SD card.
    ret = sd_card_init();
    // Check that sd card initalisation successful.
    if (ret != ESP_OK) {
        // Future led_hmi_interface updated.
        printf("sd_card_init failed: %d\n", ret);
        return -1;
    }

    // ////////////////////////////////////////////
    // Write "sd_card_init success." to unit01.txt.
    // Declare the path to write to. Create file if it doesn't exits.
    const char *write_path = "unit01.txt";
    const char *write_data = "sd_card_init success.";
    // Unit Test: sd_card_write_file.
    ret = sd_card_write_file(write_path, write_data);
    // Check that sd card initalisation successful.
    if (ret != ESP_OK) {
```

```
        printf("sd_card_write_file failed: %d\n", ret);
        sd_card_deinit();
        return -1;
    }

    // ////////////////////////////////////////
    // Read content from unit01_read.txt file.
    // Declare the path to write to.
    const char *read_path = "unit01_read.txt";
    char *read_buffer = NULL;
    size_t read_size = 0;
    // Unit Test: sd_card_read_file_dynamic.
    ret = sd_card_read_file_dynamic(read_path, &read_buffer, &read_size);
    if (ret != ESP_OK) {
        printf("sd_card_read_file_dynamic failed: %d\n", ret);
        sd_card_deinit();
        return -1;
    }

    // ////////////////////////////////////
    // Append to unit01.txt on new line.
    size_t append_data1_len = read_size + 2;
    char *append_data1 = (char *)malloc(append_data1_len);
    // Add new line.
    snprintf(append_data1, append_data1_len, "\n%s", read_buffer);
    // Unit Test: sd_card_append_file.
    ret = sd_card_append_file(write_path, append_data1);
    if (ret != ESP_OK) {
        printf("First sd_card_append_file failed: %d\n", ret);
        free(read_buffer);
        free(append_data1);
        sd_card_deinit();
        return -1;
    }
    free(append_data1);

    // ////////////////////////////////////
    // Append to unit01.txt on new line.
    const char *append_data2 = "sd_card_append_file success.";
    size_t append_data2_len = strlen(append_data2) + 2;
    char *append_data2_formatted = (char *)malloc(append_data2_len);
    // Add new line.
    snprintf(append_data2_formatted, append_data2_len, "\n%s", append_data2);
    // Unit Test: sd_card_append_file.
    ret = sd_card_append_file(write_path, append_data2_formatted);
    if (ret != ESP_OK) {
        printf("Second sd_card_append_file failed: %d\n", ret);
        free(read_buffer);
```

```
        free(append_data2_formatted);
        sd_card_deinit();
        return -1;
    }
    free(append_data2_formatted);

    // ////////////////
    // Unmount SD card
    // Unit Test: sd_card_deinit.
    sd_card_deinit();

    return 0;
}
```

Unit Test Artefacts

File Name: unit01_read.txt

File Content:

Content of unit01_read.txt

File Name: unit01.txt

File Content:

sd_card_init success.

Content of unit01_read.txt

sd_card_append_file success.

6.3.2. UT02 - Embedded Architecture of Slave Controller

Source Code Implementation

File Name: peripheral_config.h

File Content:

```
#ifndef PERIPHERAL_CONFIG_H
#define PERIPHERAL_CONFIG_H

#include "value_types.h"

typedef enum {
    DIRECTION_INPUT,
    DIRECTION_OUTPUT
} PinDirection;

typedef enum {
    SIGNAL_DIGITAL,
```

```
    SIGNAL_ANALOG
} PinSignalType;

typedef enum {
    VIRTUAL_PERIPHERAL,
    PIN_PERIPHERAL
} PeripheralType;

typedef struct PinPeripheral {
    char* PeripheralID;
    int GPIONumber;
    PinDirection PinDirection;
    PinSignalType PinSignalType;
    ValueType PeripheralDataType;
    JsonValue* PeripheralValue;
} PinPeripheral;

typedef struct {
    const char* PeripheralID;
    PinDirection PinDirection;
    PinPeripheral* pin_peripheral;
} PinPeripheralMapping;

extern PinPeripheralMapping* PinPeripheralSystemContext;
extern size_t PinPeripheralSystemContextSize;

#endif
```

File Name: peripheral_update.h

File Content:

```
#ifndef PERIPHERAL_UPDATE_H
#define PERIPHERAL_UPDATE_H

#include "esp_err.h"

typedef enum {
    UPDATE_READ,
    UPDATE_WRITE
} UpdateType;

esp_err_t peripheral_update(UpdateType update_type);

#endif
```

File Name: peripheral_update.c

File Content:

File Name: sd_card_interface.h

File Content:

```h
#ifndef SD_CARD_INTERFACE_H
#define SD_CARD_INTERFACE_H

#include "esp_err.h"

// Initialise and mount the SD card filesystem.
esp_err_t sd_card_init(void);
// Write string data to path.
esp_err_t sd_card_write_file(const char *path, const char *data);
// Append string data to path.
esp_err_t sd_card_append_file(const char *path, const char *data);
// Read file from path. Dynamic allocation in memory for dynamic file sizes.
esp_err_t sd_card_read_file_dynamic(const char *path, char **buffer, size_t
*file_size);
// Unmount the SD card filesystem and deinitialise
void sd_card_deinit(void);

#endif
```

File Name: sd_card_interface.c

File Content:

```c
// ////////////////////
// Include libraries
// External Components
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/unistd.h>
#include <sys/stat.h>
#include <errno.h>
#include "esp_vfs_fat.h"
#include "sdmmc_cmd.h"
#include "driver/sdspi_host.h"
#include "driver/spi_common.h"
// Custom Components
#include "sd_card_interface.h"

// ////////////
// Definitions
#define MOUNT_POINT "/sdcard"
```

```
// GPIO Assignment for SD Card Interface ESP32-C6-DevKit
#define PIN_NUM_MISO 6   // GPIO6
#define PIN_NUM_MOSI 4   // GPIO4
#define PIN_NUM_CLK  5   // GPIO5
#define PIN_NUM_CS   7   // GPIO7

static sdmmc_card_t *card;
static sdmmc_host_t host = SDSPI_HOST_DEFAULT();

// //////////////////
// Initalise sd card.
esp_err_t sd_card_init(void)
{
    esp_err_t ret;

    // Configure mount.
    esp_vfs_fat_sdmmc_mount_config_t mount_config = {
        .format_if_mount_failed = false,
        .max_files = 5,
        .allocation_unit_size = 16 * 1024,
    };
    // Update host configuration.
    host.slot = SPI2_HOST;
    host.max_freq_khz = SDMMC_FREQ_DEFAULT;
    // Configure SPI bus.
    spi_bus_config_t bus_cfg = {
        .mosi_io_num = PIN_NUM_MOSI,
        .miso_io_num = PIN_NUM_MISO,
        .sclk_io_num = PIN_NUM_CLK,
        .quadwp_io_num = -1,
        .quadhd_io_num = -1,
        .max_transfer_sz = 4000,
    };
    // Initialize SPI bus.
    ret = spi_bus_initialize(host.slot, &bus_cfg, SDSPI_DEFAULT_DMA);
    if (ret != ESP_OK) {
        return ret;
    }
    // Configure SD card slot.
    sdspi_device_config_t slot_config = SDSPI_DEVICE_CONFIG_DEFAULT();
    slot_config.gpio_cs = PIN_NUM_CS;
    slot_config.host_id = host.slot;
    // Mount the filesystem.
    ret = esp_vfs_fat_sdspi_mount(MOUNT_POINT, &host, &slot_config,
&mount_config, &card);
    if (ret != ESP_OK) {
        spi_bus_free(host.slot);
        return ret;
```

```
    }

    return ESP_OK;
}

// ////////////////////
// Write to sd card.
esp_err_t sd_card_write_file(const char *path, const char *data)
{
    char full_path[128];
    snprintf(full_path, sizeof(full_path), "%s/%s", MOUNT_POINT, path);

    // Open file for writing.
    FILE *f = fopen(full_path, "w");
    if (f == NULL) {
        return ESP_FAIL;
    }
    // Write data to file.
    size_t data_len = strlen(data);
    size_t bytes_written = fwrite(data, 1, data_len, f);
    fclose(f);
    // Ensure all data was written
    if (bytes_written != data_len) {
        return ESP_FAIL;
    }

    return ESP_OK;
}

// ////////////////////
// Append to sd card.
esp_err_t sd_card_append_file(const char *path, const char *data)
{
    char full_path[128];
    snprintf(full_path, sizeof(full_path), "%s/%s", MOUNT_POINT, path);

    // Open file for appending.
    FILE *f = fopen(full_path, "a");
    if (f == NULL) {
        return ESP_FAIL;
    }
    // Append data to file.
    size_t data_len = strlen(data);
    size_t bytes_written = fwrite(data, 1, data_len, f);
    fclose(f);
    // Ensure all data was written
    if (bytes_written != data_len) {
        return ESP_FAIL;
```

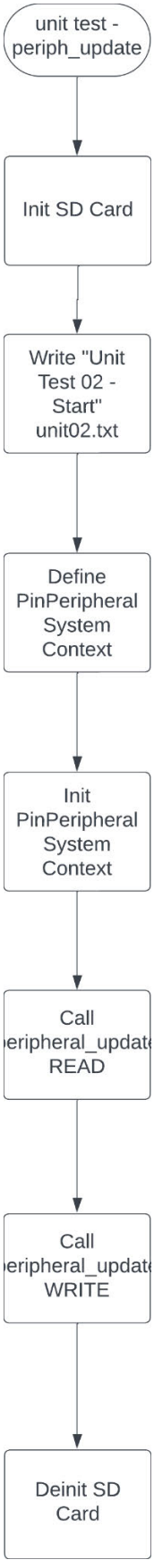
```
    }

    return ESP_OK;
}

// ////////////////////
// Read dynamic size from sd card.
esp_err_t sd_card_read_file_dynamic(const char *path, char **buffer, size_t
*file_size)
{
    char full_path[128];
    snprintf(full_path, sizeof(full_path), "%s/%s", MOUNT_POINT, path);
    // Open file for reading.
    FILE *f = fopen(full_path, "r");
    if (f == NULL) {
        return ESP_FAIL;
    }
    // Determine file size.
    if (fseek(f, 0, SEEK_END) != 0) {
        fclose(f);
        return ESP_FAIL;
    }
    long size = ftell(f);
    if (size < 0) {
        fclose(f);
        return ESP_FAIL;
    }
    *file_size = (size_t)size;
    rewind(f);
    // Allocate read buffer.
    *buffer = malloc(*file_size + 1);
    if (*buffer == NULL) {
        fclose(f);
        return ESP_ERR_NO_MEM;
    }
    // Read file content.
    size_t bytes_read = fread(*buffer, 1, *file_size, f);
    fclose(f);
    // Ensure entire file was read.
    if (bytes_read != *file_size) {
        free(*buffer);
        return ESP_FAIL;
    }
    (*buffer)[bytes_read] = '\0';

    return ESP_OK;
}
```

```
// //////////////////////
// Sd card deinitalise
void sd_card_deinit(void)
{
    esp_vfs_fat_sdcard_unmount(MOUNT_POINT, card);
    spi_bus_free(host.slot);
}
```

Unit Test Source Code Files

File Name: main.c process diagram

File Content:


unit test - periph_update
Init SD Card
Write "Unit Test 02 - Start" unit02.txt
Define PinPeripheral System Context
Init PinPeripheral System Context
Call eripheral_updat READ
Call eripheral_updat WRITE
Deinit SD Card

Unit Test Artefacts

File Name: unit02.txt

File Content:

Unit Test 02 - Start

Read: 2 = 124

Read: 10 = true

Write: 22 = false

6.3.3. UT03 – Embedded JSON Serialiser

Source Code Implementation

File Name: value_types.h

File Content:

```
#ifndef VALUE_TYPES_H
#define VALUE_TYPES_H

#include <stddef.h>
#include <stdbool.h>

typedef enum {
    TYPE_INVALID,
    TYPE_INT,
    TYPE_FLOAT,
    TYPE_STRING,
    TYPE_BOOL,
    TYPE_CHAR
} ValueType;

typedef union {
    int int_val;
    float float_val;
    const char* str_val;
    bool bool_val;
    char char_val;
} JsonValueUnion;

typedef struct {
    ValueType type;
    JsonValueUnion value;
} JsonValue;

#endif
```

File Name: json_file_seraliser.c

File Content:

```
// ////////////////////
// Include libraries
// External Components
#include "cJSON.h"
#include "esp_log.h"
#include <string.h>
#include <stdlib.h>
// Custom Components
#include "json_file_serialiser.h"
#include "sd_card_interface.h"
#include "get_schema_content.h"

static const char* TAG = "json_file_serialiser";

// Function Prototypes
static cJSON* get_terminal_value(const char* schema_name, const char*
property_name, int object_index);
static bool traverse_schema(cJSON* schema_node, const char* schema_name,
cJSON* json_node, int object_index);

char* json_schema_serialiser(const char* schema_name) {
    // Read JSON schema from SD card
    char* schema_content = NULL;
    size_t schema_size = 0;

    ESP_LOGI(TAG, "Reading schema file: %s", schema_name);
    esp_err_t ret = sd_card_read_file_dynamic(schema_name, &schema_content,
&schema_size);
    if (ret != ESP_OK) {
        ESP_LOGE(TAG, "Failed to read schema file: %s", schema_name);
        return NULL;
    }

    // Parse the JSON schema
    cJSON* schema_root = cJSON_Parse(schema_content);
    if (schema_root == NULL) {
        ESP_LOGE(TAG, "Failed to parse schema: %s", schema_name);
        free(schema_content);
        return NULL;
    }
    free(schema_content);

    // Determine the type of the root element
    cJSON* type_item = cJSON_GetObjectItem(schema_root, "type");
    if (type_item == NULL) {
        ESP_LOGE(TAG, "Schema root does not have a 'type' field");
```

```
        cJSON_Delete(schema_root);
        return NULL;
    }

    cJSON* json_root = NULL;
    if (strcmp(type_item->valuestring, "object") == 0) {
        json_root = cJSON_CreateObject();
    } else if (strcmp(type_item->valuestring, "array") == 0) {
        json_root = cJSON_CreateArray();
    } else {
        ESP_LOGE(TAG, "Unsupported root type in schema: %s", type_item-
>valuestring);
        cJSON_Delete(schema_root);
        return NULL;
    }

    if (json_root == NULL) {
        ESP_LOGE(TAG, "Failed to create JSON root element");
        cJSON_Delete(schema_root);
        return NULL;
    }

    // Start recursive traversal
    if (!traverse_schema(schema_root, schema_name, json_root, 0)) {
        ESP_LOGE(TAG, "Failed to traverse schema");
        cJSON_Delete(schema_root);
        cJSON_Delete(json_root);
        return NULL;
    }

    // Serialise the constructed JSON to a string
    char* json_string = cJSON_PrintUnformatted(json_root);
    if (json_string == NULL) {
        ESP_LOGE(TAG, "Failed to serialize JSON");
        cJSON_Delete(schema_root);
        cJSON_Delete(json_root);
        return NULL;
    }
    cJSON_Delete(schema_root);
    cJSON_Delete(json_root);

    return json_string;
}

// Recursive function to traverse the schema and construct the JSON
static bool traverse_schema(cJSON* schema_node, const char* schema_name,
cJSON* json_node, int object_index) {
    cJSON* type_item = cJSON_GetObjectItem(schema_node, "type");
```

```
    if (type_item == NULL) {
        ESP_LOGW(TAG, "Schema node does not have a 'type' field");
        return false;
    }

    const char* node_type = type_item->valuestring;

    if (strcmp(node_type, "object") == 0) {
        // Process object properties
        cJSON* properties = cJSON_GetObjectItem(schema_node, "properties");
        if (properties == NULL) {
            ESP_LOGW(TAG, "Object type node missing 'properties'");
            return false;
        }

        cJSON* property = NULL;
        cJSON_ArrayForEach(property, properties) {
            const char* property_name = property->string;
            cJSON* property_definition = property;

            cJSON* prop_type_item = cJSON_GetObjectItem(property_definition,
"type");
            if (prop_type_item == NULL) {
                ESP_LOGW(TAG, "Property '%s' does not have a 'type' field",
property_name);
                continue;
            }

            const char* property_type = prop_type_item->valuestring;

            if (strcmp(property_type, "object") == 0) {
                // Handle nested object
                cJSON* nested_object = cJSON_CreateObject();
                if (!traverse_schema(property_definition, schema_name,
nested_object, object_index)) {
                    cJSON_Delete(nested_object);
                    continue;
                }
                cJSON_AddItemToObject(json_node, property_name,
nested_object);
            } else if (strcmp(property_type, "array") == 0) {
                // Handle array
                cJSON* array_node = cJSON_CreateArray();
                // Retrieve array size using the property key
                JsonValue array_size_result = get_schema_content(schema_name,
property_name, object_index);
                if (array_size_result.type != TYPE_INT) {
```

```
                    ESP_LOGE(TAG, "Failed to retrieve array size for property:
%s", property_name);
                    cJSON_Delete(array_node);
                    continue;
                }
                int array_size = array_size_result.value.int_val;

                // Get the 'items' definition for array elements
                cJSON* items_definition =
cJSON_GetObjectItem(property_definition, "items");
                if (items_definition == NULL) {
                    ESP_LOGW(TAG, "Array property '%s' missing 'items'
definition", property_name);
                    cJSON_Delete(array_node);
                    continue;
                }

                // Iterate over array elements
                for (int i = 0; i < array_size; i++) {
                    cJSON* item_node = NULL;
                    cJSON* item_type_item =
cJSON_GetObjectItem(items_definition, "type");
                    if (item_type_item == NULL) {
                        ESP_LOGW(TAG, "Array items do not have a 'type'
field");
                        continue;
                    }

                    const char* item_type = item_type_item->valuestring;

                    if (strcmp(item_type, "object") == 0) {
                        // Handle object within array
                        item_node = cJSON_CreateObject();
                        if (!traverse_schema(items_definition, schema_name,
item_node, i)) {
                            cJSON_Delete(item_node);
                            continue;
                        }
                    } else if (strcmp(item_type, "array") == 0) {
                        // Handle nested array
                        item_node = cJSON_CreateArray();
                        if (!traverse_schema(items_definition, schema_name,
item_node, i)) {
                            cJSON_Delete(item_node);
                            continue;
                        }
                    } else {
                        // Handle terminal type within array
```

```
                        item_node = get_terminal_value(schema_name,
property_name, i);
                        if (item_node == NULL) {
                            ESP_LOGW(TAG, "Value for array '%s' index %d is
NULL", property_name, i);
                            continue;
                        }
                    }

                    cJSON_AddItemToArray(array_node, item_node);
                }

                cJSON_AddItemToObject(json_node, property_name, array_node);
            } else {
                // Handle terminal property
                cJSON* value = get_terminal_value(schema_name, property_name,
object_index);
                if (value == NULL) {
                    ESP_LOGW(TAG, "Value for property '%s' is NULL",
property_name);
                    continue;
                }
                cJSON_AddItemToObject(json_node, property_name, value);
            }
        }
    } else if (strcmp(node_type, "array") == 0) {
        // Handle array at root level
        cJSON* array_node = json_node;

        // Retrieve array size using the property key
        cJSON* title_item = cJSON_GetObjectItem(schema_node, "title");
        if (title_item == NULL) {
            ESP_LOGW(TAG, "Array node missing 'title' for property key");
            return false;
        }
        const char* array_property_name = title_item->valuestring;

        JsonValue array_size_result = get_schema_content(schema_name,
array_property_name, object_index);
        if (array_size_result.type != TYPE_INT) {
            ESP_LOGE(TAG, "Failed to retrieve array size for property: %s",
array_property_name);
            return false;
        }
        int array_size = array_size_result.value.int_val;

        // Get the 'items' definition for array elements
        cJSON* items_definition = cJSON_GetObjectItem(schema_node, "items");
```

```
        if (items_definition == NULL) {
            ESP_LOGW(TAG, "Array property '%s' missing 'items' definition",
array_property_name);
            return false;
        }

        // Iterate over array elements
        for (int i = 0; i < array_size; i++) {
            cJSON* item_node = NULL;
            cJSON* item_type_item = cJSON_GetObjectItem(items_definition,
"type");
            if (item_type_item == NULL) {
                ESP_LOGW(TAG, "Array items do not have a 'type' field");
                continue;
            }

            const char* item_type = item_type_item->valuestring;

            if (strcmp(item_type, "object") == 0) {
                // Handle object within array
                item_node = cJSON_CreateObject();
                if (!traverse_schema(items_definition, schema_name, item_node,
i)) {
                    cJSON_Delete(item_node);
                    continue;
                }
            } else if (strcmp(item_type, "array") == 0) {
                // Handle nested array
                item_node = cJSON_CreateArray();
                if (!traverse_schema(items_definition, schema_name, item_node,
i)) {
                    cJSON_Delete(item_node);
                    continue;
                }
            } else {
                // Handle terminal type within array
                item_node = get_terminal_value(schema_name,
array_property_name, i);
                if (item_node == NULL) {
                    ESP_LOGW(TAG, "Value for array '%s' index %d is NULL",
array_property_name, i);
                    continue;
                }
            }

            cJSON_AddItemToArray(array_node, item_node);
        }
    } else {
```

```
        // Handle terminal property at root level
        cJSON* value = get_terminal_value(schema_name, NULL, object_index);
        if (value == NULL) {
            ESP_LOGW(TAG, "Value for root level item is NULL");
            return false;
        }
        cJSON_AddItemToArray(json_node, value);
    }

    return true;
}

// Helper function to get terminal value
static cJSON* get_terminal_value(const char* schema_name, const char*
property_name, int object_index) {
    JsonValue result = get_schema_content(schema_name, property_name,
object_index);

    if (result.type == TYPE_INVALID) {
        ESP_LOGW(TAG, "Invalid type for property: %s", property_name ?
property_name : "(null)");
        return NULL;
    }

    switch (result.type) {
        case TYPE_STRING:
            return cJSON_CreateString(result.value.str_val);
        case TYPE_INT:
            return cJSON_CreateNumber(result.value.int_val);
        case TYPE_FLOAT:
            return cJSON_CreateNumber(result.value.float_val);
        case TYPE_BOOL:
            return cJSON_CreateBool(result.value.bool_val);
        default:
            ESP_LOGW(TAG, "Unsupported type for property: %s", property_name ?
property_name : "(null)");
            return NULL;
    }
}
```

File Name: json_file_seraliser.h

File Content:

```
#ifndef JSON_FILE_SERIALISER_H
#define JSON_FILE_SERIALISER_H

#include "cJSON.h"
```

```
// Serialize a JSON schema into a JSON file
char* json_schema_serialiser(const char* schema_name);

#endif
```

File Name: get_schema_content.c

File Content:

```
#include "get_schema_content.h"
#include "schema1_property.h"
#include "schema2_property.h"
#include "schema3_property.h"
#include "schema4_property.h"
#include "schema5_property.h"
#include <string.h>
#include "esp_log.h"

static const char *TAG = "get_schema_content";

// Define the function pointer type for retrieval functions
typedef JsonValue (*RetrievalFunction)(int object_index);

// Struct for mapping schema and property to retrieval functions
typedef struct {
    const char* schema_name;
    const char* property_name;
    RetrievalFunction function;
} SchemaPropertyMapping;

// Lookup table mapping schema-property pairs to functions
static SchemaPropertyMapping schema_property_lookup_table[] = {
    // Schema1 properties
    {"schema1.json", "string_prop", get_string_property},
    {"schema1.json", "int_prop", get_integer_property},
    {"schema1.json", "float_prop", get_float_property},
    {"schema1.json", "bool_prop", get_boolean_property},

    // Schema2 properties
    {"schema2.json", "child_string_prop", get_child_string_property},
    {"schema2.json", "child_int_prop", get_child_int_property},
    {"schema2.json", "main_float_prop", get_main_float_property},
    {"schema2.json", "main_bool_prop", get_main_bool_property},

    // Schema3 properties
    {"schema3.json", "level1_string_prop", get_level1_string_property},
    {"schema3.json", "level2_int_prop", get_level2_int_property},
    {"schema3.json", "level3_bool_prop", get_level3_bool_property},
```

```
    {"schema3.json", "level4_string_prop", get_level4_string_property},
    {"schema3.json", "level4_int_prop", get_level4_int_property},
    {"schema3.json", "level3_float_prop", get_level3_float_property},
    {"schema3.json", "root_bool_prop", get_root_bool_property},

    // Schema4 properties
    {"schema4.json", "Controllers", get_schema4_array_size},
    {"schema4.json", "string_prop", get_schema4_string_prop},
    {"schema4.json", "int_prop", get_schema4_int_prop},
    {"schema4.json", "float_prop", get_schema4_float_prop},
    {"schema4.json", "bool_prop", get_schema4_bool_prop},

    // Schema5 properties
    {"schema5.json", "UnitTests", get_schema5_UnitTests_size},
    {"schema5.json", "unit_test_name", get_schema5_unit_test_name},
    {"schema5.json", "unit_test_ID", get_schema5_unit_test_ID},
    {"schema5.json", "unit_test_state", get_schema5_unit_test_state},
    {"schema5.json", "controller_role", get_schema5_controller_role},
    {"schema5.json", "controller_ID", get_schema5_controller_ID},
    {"schema5.json", "unit_test_log_files", get_schema5_unit_test_log_files_size},
    {"schema5.json", "log_file_name", get_schema5_log_file_name},
    {"schema5.json", "log_file_version", get_schema5_log_file_version},
    {"schema5.json", "parameter", get_schema5_sub_routine_parameter},
    {"schema5.json", "parameter_value", get_schema5_sub_routine_parameter_value},
    {NULL, NULL, NULL}
};

// Function to look up the correct retrieval function based on schema and property
static RetrievalFunction lookup_function(const char* schema_name, const char*
target_property) {
    // Iterate through the lookup table to find a matching schema-property pair
    for (int i = 0; schema_property_lookup_table[i].schema_name != NULL; i++) {
        if (strcmp(schema_property_lookup_table[i].schema_name, schema_name) == 0 &&
            strcmp(schema_property_lookup_table[i].property_name, target_property) == 0) {
            return schema_property_lookup_table[i].function;
        }
    }
    return NULL;
}

// Main get_schema_content() Function
JsonValue get_schema_content(const char* schema_name, const char* target_property, int
object_index) {
    // Retrieve the function pointer based on schema_name and target_property
    RetrievalFunction retrieval_function = lookup_function(schema_name, target_property);

    if (retrieval_function == NULL) {
```

```
        ESP_LOGE(TAG, "Schema '%s' or property '%s' not recognized.", schema_name,
target_property);
        // Return a JsonValue with type TYPE_INVALID
        JsonValue error_result;
        error_result.type = TYPE_INVALID;
        return error_result;
    }

    // Call the corresponding function pointer to retrieve the property value
    JsonValue result = retrieval_function(object_index);

    return result;
}
```

File Name: get_schema_content.h

File Content:

```
// get_schema_content.h

#ifndef GET_SCHEMA_CONTENT_H
#define GET_SCHEMA_CONTENT_H

#include <stdbool.h>

// Function to retrieve content based on schema and property
JsonValue get_schema_content(const char* schema_name, const char*
target_property, int object_index);

#endif
```

Unit Test Source Code Files

File Name: main.c

File Content:

File Name: get_schema_content.c

File Content:

File Name: schema1_property.c

File Content:

File Name: schema2_property.c

File Content:

File Name: schema3_property.c

File Content:

File Name: schema4_property.c

File Content:

File Name: schema5_property.c

File Content:

Unit Test Artefacts

File Name: schema1.json.

File Content:

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Schema1",
  "type": "object",
  "properties": {
    "string_prop": {
      "type": "string",
      "description": "A string property."
    },
    "int_prop": {
      "type": "integer",
      "description": "An integer property."
    },
    "float_prop": {
      "type": "number",
      "description": "A floating-point number property."
    },
    "bool_prop": {
      "type": "boolean",
      "description": "A boolean property."
    }
  },
  "required": ["string_prop", "int_prop", "float_prop", "bool_prop"],
  "additionalProperties": false
}
```

File Name: schema2.json

File Content:

```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "title": "Schema2",
    "type": "object",
    "properties": {
      "parent_prop": {
        "type": "object",
        "properties": {
          "child_string_prop": {
            "type": "string",
            "description": "A string property inside a nested object."
          },
          "child_int_prop": {
            "type": "integer",
            "description": "An integer property inside a nested object."
          }
        },
        "required": ["child_string_prop", "child_int_prop"]
      },
      "main_float_prop": {
        "type": "number",
        "description": "A floating-point number at the root level."
      },
      "main_bool_prop": {
        "type": "boolean",
        "description": "A boolean property at the root level."
      }
    },
    "required": ["parent_prop", "main_float_prop", "main_bool_prop"],
    "additionalProperties": false
}
```

File Name: schema3.json

File Content:

```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "title": "Schema3",
    "type": "object",
    "properties": {
      "level1_prop": {
        "type": "object",
        "properties": {
          "level1_string_prop": {
            "type": "string",
            "description": "A string property at level 1."
```

```
          },
          "level2_prop": {
            "type": "object",
            "properties": {
              "level2_int_prop": {
                "type": "integer",
                "description": "An integer property at level 2."
              },
              "level3_prop": {
                "type": "object",
                "properties": {
                  "level3_bool_prop": {
                    "type": "boolean",
                    "description": "A boolean property at level 3."
                  },
                  "level4_prop": {
                    "type": "object",
                    "properties": {
                      "level4_string_prop": {
                        "type": "string",
                        "description": "A deeply nested string property at
level 4."
                      },
                      "level4_int_prop": {
                        "type": "integer",
                        "description": "A deeply nested integer property at
level 4."
                      }
                    },
                    "required": ["level4_string_prop", "level4_int_prop"]
                  },
                  "level3_float_prop": {
                    "type": "number",
                    "description": "A float property at level 3."
                  }
                },
                "required": ["level3_bool_prop", "level3_float_prop",
"level4_prop"]
              }
            },
            "required": ["level2_int_prop", "level3_prop"]
          }
        },
        "required": ["level1_string_prop", "level2_prop"]
      },
      "root_bool_prop": {
        "type": "boolean",
        "description": "A boolean property at the root level."
```

```
      }
    },
    "required": ["level1_prop", "root_bool_prop"],
    "additionalProperties": false
}
```

File Name: schema4.json

File Content:

```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "title": "Schema4",
    "type": "object",
    "properties": {
      "Controllers": {
        "type": "array",
        "items": {
          "type": "object",
          "properties": {
            "string_prop": {
              "type": "string",
              "description": "A string property."
            },
            "int_prop": {
              "type": "integer",
              "description": "An integer property."
            },
            "float_prop": {
              "type": "number",
              "description": "A floating-point number property."
            },
            "bool_prop": {
              "type": "boolean",
              "description": "A boolean property."
            }
          },
          "required": ["string_prop", "int_prop", "float_prop", "bool_prop"],
          "additionalProperties": false
        }
      }
    },
    "required": ["Controllers"],
    "additionalProperties": false
}
```

File Name: schema5.json

File Content:

```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "title": "UnitTestsSchema",
    "type": "object",
    "required": ["UnitTests"],
    "properties": {
        "UnitTests": {
            "type": "array",
            "minItems": 1,
            "items": {
                "type": "object",
                "required": ["unit_test_profile", "unit_test_controller",
"unit_test_log_files"],
                "properties": {
                    "unit_test_profile": {
                        "type": "object",
                        "required": ["unit_test_name", "unit_test_ID",
"unit_test_state"],
                        "properties": {
                            "unit_test_name": {
                                "type": "string",
                                "minLength": 1,
                                "maxLength": 100,
                                "description": "Name of the unit test."
                            },
                            "unit_test_ID": {
                                "type": "integer",
                                "minimum": 1,
                                "description": "Unique identifier for the unit
test."
                            },
                            "unit_test_state": {
                                "type": "boolean",
                                "description": "State of the unit test (true
for passed, false for failed)."
                            }
                        },
                        "description": "Profile information for a unit test."
                    },
                    "unit_test_controller": {
                        "type": "object",
                        "required": ["controller_role", "controller_ID"],
                        "properties": {
                            "controller_role": {
                                "type": "string",
                                "enum": ["Master", "Slave"],
```

```
                                "description": "Role of the controller within
the system."
                            },
                            "controller_ID": {
                                "type": "integer",
                                "minimum": 1,
                                "description": "Unique identifier for the
controller."
                            }
                        },
                        "description": "Controller details associated with the
unit test."
                    },
                    "unit_test_log_files": {
                        "type": "array",
                        "minItems": 0,
                        "items": {
                            "type": "object",
                            "required": ["log_file_name", "log_file_version",
"sub_routine"],
                            "properties": {
                                "log_file_name": {
                                    "type": "string",
                                    "minLength": 1,
                                    "maxLength": 100,
                                    "description": "Name of the log file."
                                },
                                "log_file_version": {
                                    "type": "number",
                                    "minimum": 0.0,
                                    "description": "Version number of the log
file."
                                },
                                "sub_routine": {
                                    "type": "object",
                                    "required": ["parameter",
"parameter_value"],
                                    "properties": {
                                        "parameter": {
                                            "type": "string",
                                            "minLength": 1,
                                            "description": "Name of the sub-
routine parameter."
                                        },
                                        "parameter_value": {
                                            "type": "integer",
                                            "description": "Value of the sub-
routine parameter."
```

```
                                        }
                                    },
                                    "description": "Details of a sub-routine
associated with the log file."
                                }
                            },
                            "description": "Details of a log file associated
with the unit test."
                        },
                        "description": "Array of log files related to the unit
test."
                    }
                },
                "description": "A unit test entry containing profile,
controller, and log file information."
            }
        }
    },
    "description": "Schema for unit tests."
}
```

File Name: schema1_output.json

File Content:

```
{
    "string_prop": "String Value",
    "int_prop": 1997,
    "float_prop": 19.950000762939453,
    "bool_prop": true
}
```

File Name: schema2_output.json

File Content:

```
{
    "parent_prop": {
        "child_string_prop": "Nested String Value",
        "child_int_prop": 123
    },
    "main_float_prop": 45.669998168945312,
    "main_bool_prop": true
}
```

File Name: schema3_output.json

File Content:

```
{
    "Controllers": [
        {
            "string_prop": "Controller1_String",
            "int_prop": 100,
            "float_prop": 123.44999694824219,
            "bool_prop": true
        },
        {
            "string_prop": "Controller2_String",
            "int_prop": 200,
            "float_prop": 678.9000244140625,
            "bool_prop": false
        }
    ]
}
```


File Name: schema4_output.json

File Content:


```
{
    "UnitTests": [
        {
            "unit_test_profile": {
                "unit_test_name": "UnitTest1",
                "unit_test_ID": 101,
                "unit_test_state": true
            },
            "unit_test_controller": {
                "controller_role": "Master",
                "controller_ID": 201
            },
            "unit_test_log_files": [
                {
                    "log_file_name": "LogFile1",
                    "log_file_version": 1.1000000238418579,
                    "sub_routine": {
                        "parameter": "ParameterA",
                        "parameter_value": 301
                    }
                },
                {
                    "log_file_name": "LogFile2",
                    "log_file_version": 2.2000000476837158,
                    "sub_routine": {
                        "parameter": "ParameterB",
                        "parameter_value": 302
                    }
```

```
                }
            ]
        },
        {
            "unit_test_profile": {
                "unit_test_name": "UnitTest2",
                "unit_test_ID": 102,
                "unit_test_state": false
            },
            "unit_test_controller": {
                "controller_role": "Slave",
                "controller_ID": 202
            },
            "unit_test_log_files": [
                {
                    "log_file_name": "LogFile1",
                    "log_file_version": 1.1000000238418579,
                    "sub_routine": {
                        "parameter": "ParameterA",
                        "parameter_value": 301
                    }
                }
            ]
        }
    ]
}
```

File Name: schema5_output.json

File Content:

```
{
    "UnitTests": [
        {
            "unit_test_profile": {
                "unit_test_name": "UnitTest1",
                "unit_test_ID": 101,
                "unit_test_state": true
            },
            "unit_test_controller": {
                "controller_role": "Master",
                "controller_ID": 201
            },
            "unit_test_log_files": [
                {
                    "log_file_name": "LogFile1",
                    "log_file_version": 1.1000000238418579,
                    "sub_routine": {
                        "parameter": "ParameterA",
```

```
                        "parameter_value": 301
                    }
                },
                {
                    "log_file_name": "LogFile2",
                    "log_file_version": 2.2000000476837158,
                    "sub_routine": {
                        "parameter": "ParameterB",
                        "parameter_value": 302
                    }
                }
            ]
        },
        {
            "unit_test_profile": {
                "unit_test_name": "UnitTest2",
                "unit_test_ID": 102,
                "unit_test_state": false
            },
            "unit_test_controller": {
                "controller_role": "Slave",
                "controller_ID": 202
            },
            "unit_test_log_files": [
                {
                    "log_file_name": "LogFile1",
                    "log_file_version": 1.1000000238418579,
                    "sub_routine": {
                        "parameter": "ParameterA",
                        "parameter_value": 301
                    }
                }
            ]
        }
    ]
}
```


6.3.4. Bluetooth BLE Connection Interface

Source Code Implementation

Unit Test Source Code Files

Unit Test Artefacts

6.3.5. Master Controller Configuration

Source Code Implementation

Unit Test Source Code Files

Unit Test Artefacts

6.3.6. Slave Controller Configuration

Source Code Implementation

Unit Test Source Code Files

Unit Test Artefacts

6.3.7. UT11 - Master Controller's Control Logic Engine Parser

Source Code Implementation

File Name: instruction_types.h

File Content:

```
#ifndef INSTRUCTION_TYPES_H
#define INSTRUCTION_TYPES_H

#include "value_types.h"

typedef enum {
    OP_LOAD_CONST,
    OP_AND,
    OP_OR,
    OP_NOT,
    OP_IDENTITY,
    OP_EQ,
    OP_NEQ,
```

```
    OP_GT,
    OP_GTE,
    OP_LT,
    OP_LTE,
    OP_END
} OpCode;

typedef struct {
    OpCode Opcode;
    union {
        JsonValue* OperandValue;
    } OperandData;
} Instruction;

#endif
```

File Name: control_logic_parser.h

File Content:

Unit Test Source Code Files

Unit Test Artefacts

6.3.8. UT12 - Master Controller's Control Logic Engine Interpreter

Source Code Implementation

Unit Test Source Code Files

Unit Test Artefacts

6.3.9. UT13 - Master Controller's Control Logic Engine State Manager

Source Code Implementation

Unit Test Source Code Files

Unit Test Artefacts

6.3.10. UT14 - Slave Controller's Control Logic Engine Action Dispatcher

Source Code Implementation

Unit Test Source Code Files

Unit Test Artefacts

6.3.11. Command Message Communication

Source Code Implementation

Unit Test Source Code Files

Unit Test Artefacts

6.4. Appendix D: Integration Testing

6.4.1. Master Controller and Slave Controller Communication Engine

Source Code Implementation

Unit Test Source Code Files

Unit Test Artefacts

6.4.2. Master Controller Initialisation and Slave Controller Initialisation

Source Code Implementation

Unit Test Source Code Files

Unit Test Artefacts

6.4.3. Master Controller's Configuration Engine and Control Logic Engine Integration

Source Code Implementation

Unit Test Source Code Files

Unit Test Artefacts

6.4.4. Master Controller's Control Logic Engine and Communication Engine Integration

Source Code Implementation

Unit Test Source Code Files

Unit Test Artefacts

6.4.5. Master Controller's Configuration Engine and Control Logic Engine Integration

Source Code Implementation

Unit Test Source Code Files

Unit Test Artefacts

6.4.6. Slave Controller's Control Logic Engine and Communication Engine Integration

Source Code Implementation

Unit Test Source Code Files

Unit Test Artefacts

6.5. Appendix E: System Testing

6.5.1. Operational Escape Room Purpose Testing

Source Code Implementation

Unit Test Source Code Files

Unit Test Artefacts

6.5.2. Operational Escape Room User Purpose Testing

Source Code Implementation

Unit Test Source Code Files

Unit Test Artefacts

6.6. Appendix F: Performance Testing

6.6.1. Operational Communication Coverage Performance Testing

Source Code Implementation

Unit Test Source Code Files

Unit Test Artefacts

6.6.2. Operational Communication Latency Performance Testing

Source Code Implementation

Unit Test Source Code Files

Unit Test Artefacts

6.6.3. Concurrent Slave Controller Communication to Master Controller Performance Testing

Source Code Implementation

Unit Test Source Code Files

Unit Test Artefacts

6.6.4. Concurrent Control Logic Evaluation Performance Testing

Source Code Implementation

Unit Test Source Code Files

Unit Test Artefacts