University of Southern Queensland

School of Engineering

Investigation of Web-Based Technologies in the Development of Full-Stack, Small to Mid-Sized Distributed Industrial Control Systems

A dissertation submitted by

S. Carr

in fulfilment of the requirements of

ENG4111 Professional Engineer Research Project

towards the degree of

Bachelor of Engineering (Honours)(Instrumentation & Control)

Submitted: November, 2024

Abstract

Distributed Industrial Control Systems employ networks to connect distant controllers and allow seamless integrated control and monitoring of processes. Two critical aspects of these systems are their horizontal functionality, with communications based on real-time control, and secondly their respective network design to achieve this (Selişteanu, Roman, Şendrescu, Petre & Popa 2018). The majority of these systems are fully proprietary and come with high purchase costs, with large-scale systems reaching into the tens of millions of dollars.

Comparatively, improvements in low-cost, embedded controller performance have seen these devices gain more widespread use for a range of applications. As these systems improve, low-cost does not have to be synonymous with low-reliability, or low-functionality. A body of research currently exists in which embedded systems have been used for developing remote data systems. An area that is understated in such research is in the application of connected embedded systems to form distributed control and data networks.

This dissertation shows that embedded systems, in conjunction with web-based programming languages, can be used to deliver a reliable distributed control network. Such a network can be developed to offer real-time communication with data storage and visualisation capabilities. The developed system demonstrated how simplified and customisable integration of sensors could be achieved with interfacing, data processing and storage software which had the capability to be expanded for large numbers of distributed sensors across a network.

By developing NodeJS applications in conjunction with a WebSockets Protocol communication network, a single Server, dual Client architecture was built, capable of fast, customised communication and control. The system displayed how specific time thresholds could be met and monitored, through use of watchdog and time-stamping applications and also provided data management through a connected server database system. This project also documented approaches to manage time-dependent operations using web-based, event-driven, software callback methods.

These outcomes were important to demonstrate the potential for a differing approach to control system development. They confirmed that by using open-source, web-based programming languages and runtimes, architectures can be developed which allow real-time constraints to be managed all while simplifying system integration. In doing so, such systems could offer a high level of reliability in operation and be looked upon to potentially replace legacy devices which have been slower to adapt in software terms.

University of Southern Queensland

Faculty of Health, Engineering & Sciences

ENG4111/2 *Research Project*

Limitations of Use

The Council of the University of Southern Queensland, its Faculty of Health, Engineering & Sciences, 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 Faculty of Health, Engineering & Sciences 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.

Dean

Faculty of Health, Engineering & Sciences

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.

S. Carr

[redacted]

Acknowledgments

I would like to offer my sincere appreciation to my academic supervisor, John Leis. Thank you for your highly productive communication and timely guidance over the course of this project, from start to finish you were always available to assist.

This project and my studies could not have been accomplished without the tireless support of my amazing wife Casandra along with my three children Violet, Daisy and Austin. Thank you for your support over the years and to my extended family who also greatly assisted over this time.

Finally, thank you to all the UniSQ faculty staff who have assisted me over my years of study.

S. Carr

Contents

Abstract **i**

Acknowledgments **vii**

List of Figures **xvii**

List of Tables **xxiii**

Chapter 1 Introduction **1**

1.1 Chapter Overview 1

1.2 Motivation 2

1.3 Project Aim 2

1.4 Research Objectives 4

1.5 Overview of the Dissertation 5

Chapter 2 Literature Review **7**

2.1 Chapter Overview 7

2.2 Modern Technology, Growth and Applicability 8

2.2.1 Node.js – An Event-Driven JavaScript Runtime 8

2.2.2 Node Package Manager (NPM) 11

2.2.3 Express JS 12

2.2.4 Embedded JavaScript Templating (EJS) 12

2.2.5 Additional Programming Software and Techniques 13

2.2.6 The WebSockets Protocol 14

2.3 Existing State of Technology 18

2.3.1 PLC's and the IEC61131-3 Standard 19

2.3.2 Distributed Systems and the IEC61499 Standard 21

2.3.3 Developments and Trends 22

2.3.4 Protocols used, Data Mapping and Interfacing Requirements. . . . 23

2.4 Similar Approaches, Implementations and Benchmarks 24

2.5 Chapter Summary 32

Chapter 3 Methodology **33**

3.1 Chapter Overview 33

3.2 Introduction 34

3.3 Project Documentation 34

3.4 Risk Management 35

3.5 Project Breakdown and Timeline 35

3.6 Hardware 37

3.6.1 Distributed Control System Architecture 37

3.6.2 Raspberry Pi 5 38

3.6.3 Raspberry Pi Zero W 39

3.6.4 Raspberry Pi ADC Module 40

3.6.5 PWM 3.3V/5V Voltage Converter Board 41

3.6.6 Ethernet Unmanaged Switch 41

3.6.7 24 Volt DC Power Supply Unit 42

3.6.8 Miscellaneous Equipment 42

3.7 Software 43

3.7.1 Raspian and Ubuntu/Linux OS 43

3.7.2 NodeJS & NPM 43

3.7.3 ExpressJS & EJC 44

3.7.4 MySQL 45

3.7.5 MS Visual Studio Code 45

3.7.6 Putty 46

3.7.7 Git and GitHub 46

3.8 Details of Proposed Methods and Testing 47

3.8.1 Individual Control Server Applications 48

3.8.2 Communication Network 49

3.8.3 Central Server Scheduling and Timekeeping 49

3.8.4 Database Setup and Operation 50

3.8.5 Human-Machine-Interface (HMI) 50

3.8.6 Class-Based Modules for Control and Data Handling 51

3.8.7 Hardware Interfacing 52

3.8.8 Further Sequential Testing 52

3.9 Records and Data Analysis 52

3.10 Chapter Summary 54

Chapter 4 Functional Design 55

4.1 Chapter Overview 55

4.2 Hardware Design and Construction 56

4.2.1 Raspberry Pi Five - Main Server 57

4.2.2 Raspberry Pi Zero Units - Distributed Clients 58

4.2.3 Hardware - Material Take-Off (MTO) 60

4.2.4 Test Equipment 61

4.3 Software Design and Development 62

4.3.1 Pi Five Server - Program Architecture 62

4.3.2 Pi Zero Clients - Program Architecture 66

4.3.3 Network Communications and WebSockets Operations 68

4.3.4 Sequential and Time-Based Control within Server / Client Applications 77

4.3.5 Database Operations 83

4.3.6 Model-View-Controller - HMI display 87

4.3.7 System Information and Data Monitoring 93

4.4 Chapter Summary 95

Chapter 5 Results and Discussion **97**

5.1 Chapter Overview 97

5.2 Preliminary Test Results 98

5.2.1 General Purpose Input Output Testing 98

5.2.2 Ping Testing of Ethernet Connections 101

5.2.3 Basic WebSockets Connection 102

5.2.4 Terminal and Console Output Logging 103

5.3 Detailed Test Results 106

5.3.1 Time Logging of Data & Execution Timekeeping 107

5.3.2 Data Management and Storage 109

5.3.3 Communications Reliability 111

5.3.4 Human-Machine Interface Development Results 113

5.3.5 Sequential and Asynchronous Event Operation 114

5.3.6 Analog & Pulse-Width Modulation - Input / Output Testing . . . 116

5.4 Chapter Summary 118

Chapter 6 Conclusions and Further Work **119**

6.1 Chapter Overview 119

6.2 Conclusions 120

6.3 Achievement of Research Objectives 122

6.4 Further Work 123

6.5 Chapter Summary 124

References **125**

Appendix A Project Specification **131**

Appendix B Risk Assessment **138**

Appendix C Ethical Clearance **140**

Appendix D Supporting Information **142**

D.1 Introduction to this Appendix 142

D.2 Further Technical Information and Specifications 143

D.3 Installations and Configurations 145

D.3.1 Operating systems, Installations and Runtime 145

D.3.2 NodeJS Installation and Application Initialisation 148

D.3.3 MySQL Installation and Database Initialisation 151

D.3.4 ADC Pi interface card, setup and testing 154

D.3.5 Pulse Width Modulation Output, setup and testing 156

D.4 Software Code Files 159

D.4.1 server.js code 159

D.4.2 client1.js code 162

D.4.3 client2.js code 165

D.4.4 serverSocket.js code 167

D.4.5 clientSocket.js code 171

D.4.6 watchdog.js code 175

D.4.7 system.js code 178

D.4.8 date.js code 181

D.4.9 database.js code 183

D.4.10 devices/client1iodata.js code 188

D.4.11 devices/client2iodata.js code 193

D.4.12 devices/clientAnalog.js code 198

D.4.13 devices/clientDigital.js code 203

D.4.14 devices/clientSendData.js code 205

D.4.15 devices/serverFormatData.js code 207

D.4.16 controllers/error.js code 209

D.4.17 controllers/hmi.js code 211

D.4.18 models/devices.js code 213

D.4.19 public/css/main.css code 215

D.4.20 public/images/mockPT.png 217

D.4.21 routes/hmi.js code 219

D.4.22 views/hmi/404.ejs code 221

D.4.23 views/hmi/index.ejs code 223

D.4.24 views/includes/head.ejs code 225

D.4.25 views/includes/end.ejs code 227

D.4.26 pwm.js code (Tested as individual Node executable) 229

D.4.27 adcpi.js code (Tested individually and within updateIO() function) 231

List of Figures

2.1 Node.js Runtime and engine overview. (Nagarajan 2023). 9

2.2 Overview of control flow for an asynchronous promise function (Kadlecsik 2021). 11

2.3 Display of the fundamental architecture for an MVC system (Hernandez 2021). 14

2.4 WebSockets against standard polling techniques (Mbed. 2024). 15

2.5 Detailed plot of a Server / Client message distribution using Websockets (Murley, Ma, Mason, Bailey & Kharraz 2021). 16

2.6 A simplified Distributed Control System Architecture example (Gillis 2023). 19

2.7 Representation of the 5 recognised IEC61131-3 language approaches (Budimir 2018). 21

2.8 System Model overview of the IEC61499 Standard (IEC61499 2024). . . 22

2.9 Distribution of Industrial Protocols and growth showing ethernet weighting (Fluke 2024). 24

2.10 Abstraction layer overview common to software engineering (Racchetti, Fantuzzi, Tacconi & Bonfè 2015). 26

2.11 Command-based architecture (Vargas, Sánchez, Jara, Candelas, Torres & Dormido 2011). 27

2.12 Event-Driven Architecture example with standard central broker (Shabani 2018). 28

2.13 Overview of an integration of Industry 4.0 systems (Bellini, Cenni, Mitolo, Nesi, Pantaleo & Soderi 2022). 29

2.14 Example of a Node-RED workflow into an IoT app (Musings. 2020). . . . 30

2.15 Developed framework for assessing an Industrial Internet Platform (Li, Zhou, Zhang, Liu & Li 2021). 31

3.1 Timeline of critical events as part of the research project. 36

3.2 Overview of the Distributed Control System Architecture / Topology . . . 37

3.3 Raspberry Pi 5 Embedded Board (Raspberry-Pi. 2023) 39

3.4 Raspberry Pi Zero W Embedded Board (Core-Electronics. 2024) 40

3.5 Raspberry Pi ADC Module (PiHut. 2024) 40

3.6 PWM 3.3V/5V Voltage Converter Board (Amazon. 2024) 41

3.7 TP Link 5-Port Ethernet Switch (TP-Link 2024). 42

3.8 24 Volt DC Power Supply (ebay. 2024*b*) 42

3.9 An open terminal instance with request to download nodeJS 44

3.10 Example node application source code with dependency listing 44

3.11 The basic intended software topology for the Distributed Control System. 47

4.1 Block Diagram of all Power, Communications and accessory connections. 56

4.2 Image of the hardware connections to the Pi Five unit. 57

4.3 Image of the hardware connections to one of the Pi Zero units. 58

4.4 Image of the final system hardware setup. 59

4.5 Server software sequence flowchart. 63

4.6 Overview of the system dependencies for the Server architecture. 65

4.7 Client software sequence flowchart. 66

4.8 High-Level WebSockets network diagram 68

4.9 Pi Five WebSockets Server software sequence flowchart. 70

4.10 Pi Zero W WebSockets Client software sequence flowchart. 74

4.11 Communications Network Watchdog software sequence flowchart. 76

4.12 updateIO() device scan software sequence flowchart. 79

4.13 aiDevice() class software flowchart. 81

4.14 MySQL Database and NodeJS server application integration flowchart. . . 84

4.15 myDB() function screenshot. 85

4.16 ioToDB() function screenshot. 86

4.17 VS Code folder and file MVC structure. 87

4.18 The HMI homepage router responsible for calling the index page and assigned controller 88

4.19 The HMI homepage controller 89

4.20 The HMI error page controller 89

4.21 The HMI Device Model 90

4.22 EJS includes ‘Head’ file 90

4.23 HMI homepage ‘views/hmi/index.ejs’ file 91

4.24 HMI Model-View-Controller Software Flowchart. 92

4.25 The ‘createSysMap()’ function 93

4.26 The ‘sysInfo()’ function 94

5.1 Raspberry Pi Zero W GPIO Header Layout (Pi4J. 2019) 99

5.2 Pi Zero W GPIO test circuit. 100

5.3 NodeJS GPIO test code. 100

5.4 Pi Five WebSockets Server connectivity and message confirmation. 102

5.5 Pi Zero W WebSockets Client connectivity and message confirmation. . . 103

5.6 Code execution with added test points. 104

5.7 Promise testing with custom message confirmations of sequence. 104

5.8 Client 1 watchdog timestamp and custom message confirmations. 105

5.9 client1.js execution time monitoring. 107

5.10 MySQL Workbench Visual data representation. 109

5.11 Database table data exported in CSV format. 110

5.12 Basic HMI home screen rendered using MVC framework. 113

5.13 High High delay on analog device using promise callbacks. 114

B.1 UniSQ Risk Assessment for the project. 139

D.1 Image of the Etcher process to flash the Micro-SD’s with OS image. . . . 146

D.2 IPv4 configuration with ifconfig confirmation of settings. 147

D.3 Enabling configuration settings via preferences tab. 147

D.4 Intialisation of NodeJS application using npm package manager command. 149

D.5 Visual Studio Code Server application and file / folder structure. 150

D.6 Terminal login to MySQL program. 151

D.7 Connection to MySQL database via Workbench 8.0. 153

D.8 Code used to test ADC Pi interface with NodeJS 154

D.9 Program scaled output whilst scanning input. 155

D.10 Test equipment with readings during tests 155

D.11 PWM signal on Picoscope during output testing. 156

D.12 Unsatisfactory PWM signal displayed on Picoscope. 157

D.13 PWM board wiring diagram. 157

List of Tables

4.1 Material Take-Off for project equipment 60

4.2 Test equipment used within project 61

5.1 Ping Connection and Latency results across the network 101

5.2 Execution times in milliseconds for NodeJS programs and functions. . . . 108

5.3 Results of analog scan and scale testing 116

5.4 Results of Pi Zero W PWM output testing 117

Chapter 1

Introduction

1.1 Chapter Overview

This chapter provides an introduction to the project, describing the motivation, project aim and research objectives. It also provides an overview of each chapter, offering a brief description of each.

1.2 Motivation

Industrial control is the technology stream tasked for ensuring safe, efficient, and productive manufacturing and processing operations can take place in the modern world. For systems which are spatially separated a distributed architecture is the chosen method for allowing control and communications between assets or sections of plant.

Dependent on the size of the installation, the complexity and cost can vary dramatically, with higher end systems for large facilities costing into the ten's of millions of dollars. Such systems can host a plethora of advanced features and use proprietary technology which quickly moves out of the range for smaller operational budgets. The focus of this dissertation was limited to the review of legacy approaches for small to mid-sized distributed systems.

Over time, small embedded controllers have become more powerful in processing and interfacing capabilities (Fang & Fu 2011). Such controllers have also become able to host highly-flexible software run-times which have libraries for development and allow custom coded programs to be developed for added functionality. Web-Based technologies in parallel, have allowed high performance, scalable applications to be developed with real-time communications built-in (Branch & Bradley 2006).

1.3 Project Aim

With consideration to the above factors, the project aim was to assess whether modern Web-Based software technologies and communication protocols, in conjunction with lower-cost embedded hardware solutions, could offer competitive solutions and benchmark performance to their legacy industrial counterparts.

While the complete development of a replacement system was beyond the scope of this project, selected elements, deemed critical were researched. The project aim was to determine if solutions reflective of these elements could be developed, which demonstrated the capabilities of Web-Based embedded solutions to provide industrial-level performance.

Industrial-level performance relates to the ability to establish and persistently maintain communications between server and clients at a low millisecond transmission rate. Further to this, connection monitoring for both loss of connection, and latency issues, was a requirement to ensure the communication links were persistent and reliable.

Once these reliability and performance metrics were able to be proven, additional functionality was developed to showcase the potential further benefits of using such technologies. This included the ability to create an integrated database allowing streamlined and remote access to data from systems and devices, a Human-Machine Interface display that could also integrate directly with the server application, and class-based software which could closely match specific functionality of standard legacy controller logic.

This additional functionality aligns with the below points which were listed as key components of legacy distributed control system architectures. They are further explored and detailed in Section 3.8.

- Individual Control Server Applications
- Communications Network
- Central Server Scheduling and Timekeeping
- Database setup and operation
- Human-Machine-Interface (HMI)
- Class-Based Modules for control and data handling
- Further Sequential Testing

1.4 Research Objectives

The research objectives which were initially defined within the Project Specification in Appendix A are listed below:

- Conduct a detailed review of the key components in mid-sized control systems and detail how the selected software can be utilised to develop both theoretical, and where possible, practical solutions of each equivalent section.
- Implement and test performance of a WebSockets based communication layer over multiple nodes to allow distributed data channels and control to be realised.
- Review and assess how an implementation of the NodeJS server runtime can be structured to meet potential schedule deadlines and deliver reliable core control system operation.
- Develop sound, re-useable, modular code and class-based systems for any control system architectures created.

1.5 Overview of the Dissertation

This dissertation is organized as follows:

Chapter 2 - Literature Review: Develops the fundamental ideas for the project. Reviews both legacy and modern technologies, with examples of documented approaches which rival traditional control and distributed systems.

Chapter 3 - Methodology: Provides a detailed outline of how selected hardware and software approaches could be used to develop solutions. Further, it lists the performance attributes used for assessment during the design stage of this project.

Chapter 4 - Functional Design: Reviews the hardware design and the software development cycle including the structure and program flows, dependencies and system components.

Chapter 5 - Results and Discussion: Reviews the test results delivered during development and operation of the distributed control system. Offers final stage results of the running system.

Chapter 6 - Conclusions and Further Work: Summarises the dissertation, states the achievement of outcomes and details further work to be undertaken.

Chapter 2

Literature Review

2.1 Chapter Overview

The following section assists to develop the fundamental ideas on which this project was based. It reviews the existing and potential technologies implemented for industrial control systems with distributed architectures, and in doing so showcase key characteristics, benefits, detractors and required performance benchmarks. The section is separated into three major sub-sections:

1. Modern Technology, Growth and Applicability: Explores modern web-based technologies, specifically the JavaScript based runtime Node.js and WebSockets communication protocol. Review of operation, key characteristics, overall benefits, and considerations are covered.

2. Existing State of Technology: Provides a background of legacy approaches and developments, including technologies and software. Specifically, the IEC61131-3 programming language formats, IEC61499, the Distributed Automation Architecture Standard and Industrial Protocols under IEC61158/IEC61784.

3. Similar Approaches, Implementations and Benchmarks: This sub-section focusses on previous examples which are similar and relevant to the elements of this project. Offering insight into methods, outcomes, and learnings. It also provides references for performance-based targets relating to distributed industrial control systems of this nature and how and where they were applied for the project.

2.2 Modern Technology, Growth and Applicability

This project aimed to employ web-based programming software to replicate aspects of normal control system function, prove that reliable operation can be met and maximise the ability of such technology to ease access to data and create flexible models in the process. This was to be achieved using differing techniques to standard controller and PLC language approaches, as explored above in relation to the possibility of hybrid time and event driven systems. One of the core languages which has seen high use for event-driven operation in web-based systems is Node.js (Ancona et al. 2017). This was been selected as the base runtime system for the application due to some highly relevant operating traits which are detailed below.

2.2.1 Node.js – An Event-Driven JavaScript Runtime

There are many languages and programming stacks used for web-development with each offering benefits and specific characteristics for developers to utilise. One specific programming runtime which has seen a great deal of growth since its creation by Ryan Dahl in 2009 is Node.js. Originally designed by Dahl to simply allow the dynamic display of a progress bar within a larger production grade application, the runtime used callbacks from the server to the browser to provide the updated completion data for uploaded files (Shah & Soomro 2017).

Node.js is based on the high-performing Google Chrome V8 JavaScript Engine, this is built from the ground up using C++ with the resulting runtime using the programming language JavaScript, introduced in 1995. The Node.js runtime is efficiently designed to run as an event-based, not thread-based architecture and can run on multiple operating systems such as Windows, Linux and Mac OS. It can also run within the Google Chrome Browser, embedded within C++ programs or standalone (Radix 2024).

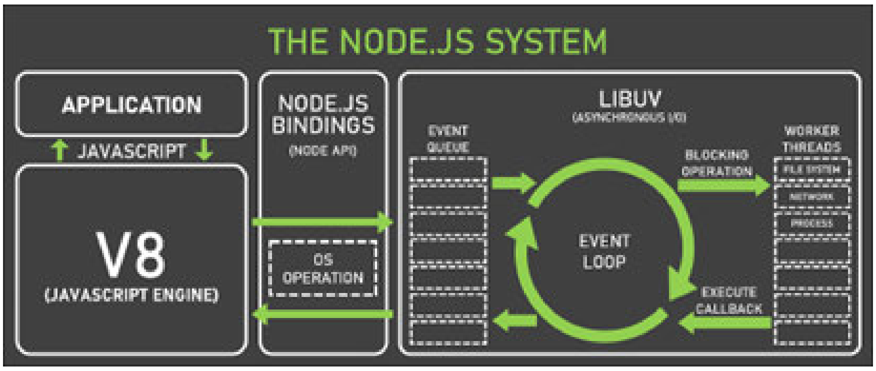


Figure 2.1: Node.js Runtime and engine overview. (Nagarajan 2023).

Node.js provides a lightweight structure which can be used to scale for large numbers of users when implemented correctly, a key reason for its high growth. As of 2023, Node.js was listed as a framework used by over forty percent of developers worldwide (Statista 2022) and it has seen use in applications for some highly notable corporate entities. Node.js has been created as a single-threaded, non-blocking, event-loop runtime. What this means when implemented correctly, is that the main loop runs continuously without any single operation causing the loop to hold for extended periods before moving to the next operation.

Comparative systems could see an equivalent operation cause the program to wait for data to be returned or a specific algorithm result, a term coined 'blocking' in software design. This condition results in delays which can quickly cause issues for highly scaled systems with many concurrent users. In Node.js the system is non-blocking, which is achieved by running a single, main event-loop thread, which is continually responding to individual event requests and responses, with concurrent connections sometimes into the millions (Shah & Soomro 2017).

The method which allows the main thread to run so efficiently when scaled and loaded heavily is the fact that it can separate out tasks and allocate these to worker threads held within an overall pool. This delivers an adaptable running process which can ensure that in general no individual action will create a performance issue across the application. The operation of the overall application depends on various implementation factors, most notably the way in which the event-based model is constructed via the event-loop structure. This in turn is dependent on the LIBUV multi-platform C library as shown in the above Figure 2.1, which is responsible for the event loop itself and thread pool functionality (Patrou et al. 2021).

The event loop thread is the main runtime thread which initialises the application and manages the associated requests and responses. The architecture maintains an event queue and executes an important aspect of Node.js, *callbacks*, which are the system responses for completed operations. The I/O network operations are executed with non-blocking inter-process communication sockets, and actions associated with the file system and various other process operations are sent to be executed within a separate worker thread pool.

The worker pool can be used for system calls between JavaScript and the C++ system operations and although such operations add additional time latency, this structure importantly allows the main thread to continue progression (Patrou et al. 2021). Initially, there are four threads available which can be expanded up to 1024 as required along with memory allocation.

A point of note is that Node.js is not suited for high-computational CPU loads which run a single process, this is not what Node.js has been developed for and results in poor performance and overall delay in the event-loop (Shah & Soomro 2017). The process can slow considerably when synchronous activities are required which compound loading on the system, or many long running functions combine to adversely affect performance. Examples include AI and machine learning computations and while there are ways to design the core program to reduce system resource depletion, the developer must understand how the tasks and program operations will be allocated when including such heavy computational operations within the scope of an application.

Node.js was developed to allow a system to respond to large numbers of smaller, individual events which are sent to the event-loop and do not individually require large amounts of overall computation. Examples are requests and user interactions on webpages which can quickly be resolved allowing a fast, efficient loop to operate and maintain performance. This philosophy needed to be factored when developing the control system solution to ensure the methods of implementation did not conflict with the overall system structure.

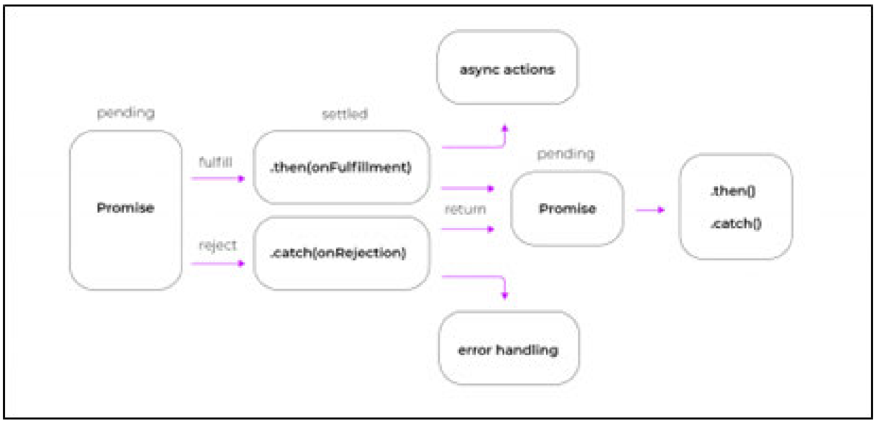


Figure 2.2: Overview of control flow for an asynchronous promise function (Kadlecsik 2021).

When requiring sequential operations within a Node.js application there are multiple methods that can be utilised to ensure the system flow is maintained. As mentioned, callbacks are an inherent operation within the programming flow, further to this, *promises*, which are an improved callback method, can be utilised to create execution dependencies within the asynchronous operations of the application. Utilisation of such functions within the application allow for sequential operations and ordered execution, accordingly this is a critical capability within the otherwise event-driven process flow. Such functions can be structured within a control system application to ensure if an ordered process flow is required, it can be correctly executed.

Another positive attribute of the Node.js runtime is that it was developed to allow a unified programming language to run both server and client-side operations. This is another major characteristic which resonated with the requirements of this project to minimise language changes and reduce added complexity to development and data handling.

2.2.2 Node Package Manager (NPM)

One significant positive characteristic of Node.js is the large user base and associated open-source libraries available for use and inclusion within developed projects. As part of the large Node.js ecosystem, Node Package Manager is the biggest single language repository on earth with over 2.1 million packages listed as of September 2022 (Nodejs.org. 2024). The NPM repository allows for rapid development and re-use of existing modules by developers which is a contributing factor for why Node.js is so popular in the web landscape.

The package manager holds a variety of packages which are accessed via the command line, easily allowing fast building of core functions such as front-end web application frameworks, mobile applications, routers and various other individually developed packages for select operations (Npmjs. 2024*a*). There was a need to call upon various modules as part of this project to assist in the development of some of the fundamental aspects of the control architecture. While the overall system was a novel design, the base software structure and various libraries were utilised to add functionality, these are documented and referenced where included.

2.2.3 Express JS

Express JS is a middleware software application framework relied upon for streamlining and simplifying development methods. The framework sits above the existing Node.js functionality to assist in reducing laboursome coding requirements especially when working with HTTP and application routing tasks. The software manager uses specific modules which exist within the overall functional framework to enhance the readability, and flow of operation of application programming. This is achieved by assisting with parsing of data (handling functions) and providing a rich set of efficient coded functions for common tasks within application development (Expressjs. 2024).

2.2.4 Embedded JavaScript Templating (EJS)

EJS is a templating language which provides the ability to generate HTML via normal JavaScript by using leading and trailing markers within the coded sections. The language allows fast, cached scripts to be embedded linking functionality across an application easily and efficiently. It also provides high-level error checking with line-marking and exceptions further assisting in streamlined development (Eernisse 2020).

2.2.5 Additional Programming Software and Techniques

Further to the above specified software packages and suites, the project utilised additional software for various purposes. HyperText Markup Language (HTML) and Cascading Style Sheets (CSS) which are core web-based languages were employed to assist in visual display and HMI based functions. A version of databasing suite named MySQL was installed and used to assist in data storage and management within the terms of the project. It is one of the most used open-source databases in the world and is highly compatible with the Raspberry Pi 5 which has been selected for use as the central embedded server for the project, based on cost, accessibility, and performance.

One added technique which was be investigated as part of the project software solution was the implementation of a Model-View-Controller (MVC) architecture for the main application structure. This is a system of design often employed for web-based applications and sectionalises the overall functions to ensure a modular system interface which is said to create a 'separation of concerns' (Liu & Wang 2012).

The Model component is where the logic for data access and retrieval from a database is held. The View component is the separated code which serves the function of user interaction, allowing communication with the controller to provide the actions called upon by the user such as keyboard entries and requests. The Controller acts as an intermediary, tasked with joining the Model and View sections through specific functional operations. This structure allows for a more defined, component like design which is good for scalability, reuse, and readability.

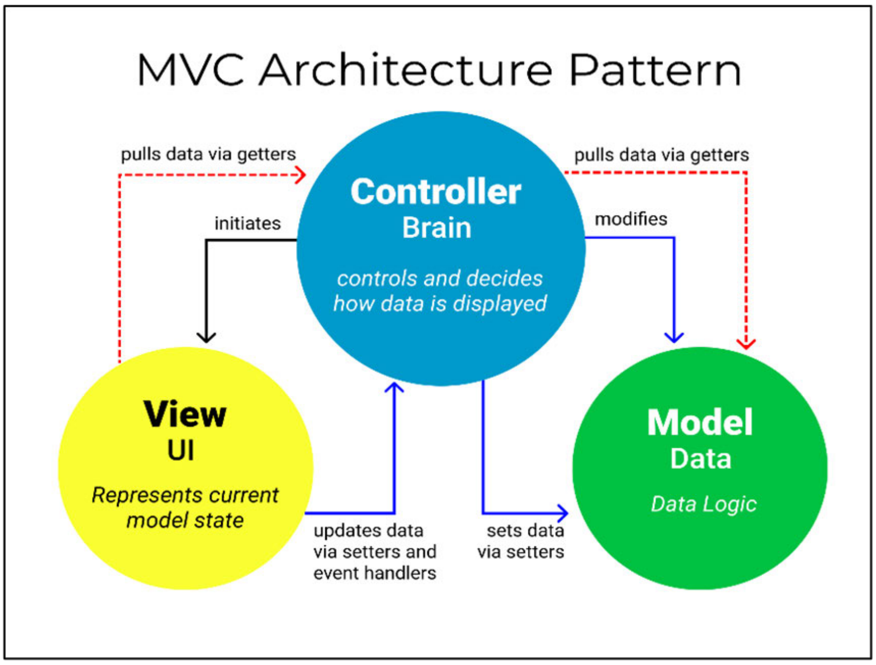


Figure 2.3: Display of the fundamental architecture for an MVC system (Hernandez 2021).

2.2.6 The WebSockets Protocol

The WebSockets Protocol is an application layer protocol which sits on top of the TCP/IP suite. It has seen usage officially from 2011 onward (Murley et al. 2021) and is used globally by some high-profile companies to deliver production grade, reliable, commercial applications. Having been closely developed for Web-Based applications it has many characteristics which assisted in the development of a fast and robust distributed control architecture for this project.

In relation to the chosen programming runtime of Node.js and JavaScript, WebSockets which can also perform non-blocking operations has been shown to be highly compatible with excellent scalable results when combining the two technologies (Tomasetti 2021). As the WebSockets Protocol itself is built above TCP/IP it is already tailored for flexibility in developing individual data streams which can take on various object-based structures at the application level. When assessed on time latency in operation, one study has shown that when compared to the 2nd layer raw TCP/IP connection, a WebSocket connection took 3.7 times longer to establish the connection. While this is a latency cost, it is minimal considering the protocol sits two layers higher on the OSI-model on the transport layer, responsible for the message packet structure afforded by the protocol.

It was also found that there was an approximate sixty percent higher computational workload for the application processing the data. This increase coupled with the longer establishment time could be seen as detractors, but given the higher-level structure, the protocol does not significantly load the connection when considering the lower, raw layer 2 TCP/IP data stream (Skvorc et al. 2014).

The major characteristic of WebSockets which sees it employed for chat and streaming applications globally is the ability to transport real-time data between communication nodes by implementing asynchronous and full-duplex communication. Most WebSockets Protocol suites make use of event-driven callback monitoring and control and have a host of additional functionality built into the libraries which can be imported such as Socket.io, a popular choice for web applications.

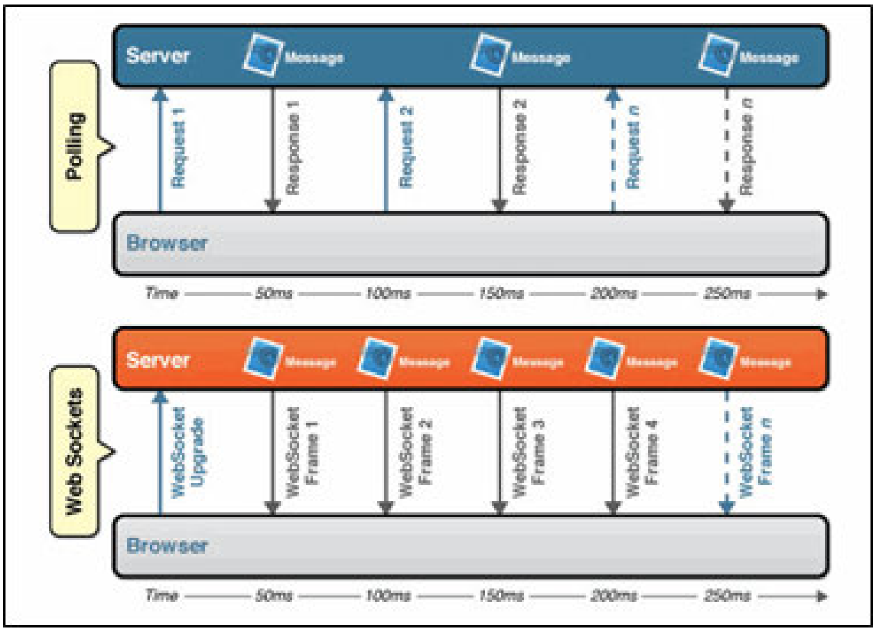


Figure 2.4: WebSockets against standard polling techniques (Mbed. 2024).

Before WebSockets the main method of communication over HTTP was a long-polling technique which would see a server hold a request until the return data was ready to send, creating a number of computational overheads and delays. The long-polling technique would reduce the number of requests and latency by continually renewing a series of ongoing requests instead of closing them off however WebSockets improves further by creating a fully persistent connection. This approach leads to greater efficiency and latency reduction throughout the connection lifetime (Qveflander 2010).

To ensure that an industrial communication protocol can provide reliable operation it must be shown that it can meet speed and error handling targets. One performance-based study labelled 'Research of Web Real-Time Communication Based on WebSocket' (Liu & Sun 2012), explored connection capabilities of WebSockets showing positive outcomes. The study used a C library to assess connections with asynchronous transmission against a standard HTTP connection with results showing a tenfold improvement over traffic and network delays.

As previously detailed WebSockets uses an event-driven call structure which heavily suits request-response activities but characteristically does not need the client to be the initiator of a request. This is where Web-Based applications commonly see a server led messaging routine interact with users much easier.

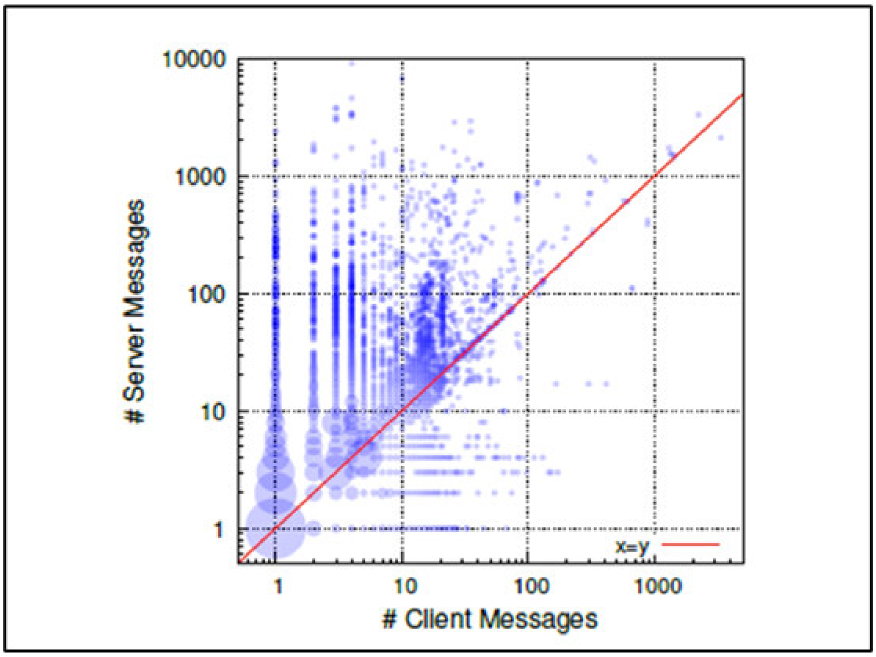


Figure 2.5: Detailed plot of a Server / Client message distribution using Websockets (Murley et al. 2021).

A research paper titled 'WebSocket Adoption and the Landscape of the Real-Time Web' (Murley et al. 2021) carried out a review of websites and their activities involving WebSocket implementations. It was found that in one case a website successfully sent over 31300 messages originating from the server at a frequency of approximately 1.4 milliseconds showcasing just how fast and efficient the protocol can be over a distributed communication channel.

A critical aspect of any protocol is the ability to deliver a secure, uninterrupted channel for constant data flow. Given that WebSockets is predominantly a Web-Based protocol it has seen some rigorous security methods employed to keep sensitive user data protected and reduce the likelihood and ability of common cyber-attack vectors. The WebSockets RFC 6455 details methods to deliver the intended use and structure of a robust implementation however this is not always achieved for numerous reasons, many relating to developer experience and decision-making.

As WebSockets is a choice for fast, accessible, and low-overhead communication, the other critical aspects of securing the protocol can be neglected due to more complex implementation factors, overall data transfer speed and performance reductions. Research by Jusso Karlström in the paper ‘The WebSocket Protocol and Security: Best Practices and Worst Weaknesses’ describes how a certain lack of explicit direction within the RFC 6455 can lead to failure to secure the protocol on occasions and issues with overall standardisation.

The paper does however continue to note that growing web adoption has led developers to increase the standard and measures for encryption and other factors relating to security. The researcher states ‘All things considered, the WebSocket Protocol can be used as a secure, efficient and low overhead solution for the web applications that need real-time and two-way communication,’ (Karlström 2016). Some points of note are methods of handshaking the connection, untrusted certificate detection, header verification and data frame checks, some relevant elements of industrial data exchange practices.

Harri Kuosmanen carried out a review titled ‘Security Testing of WebSockets’ (Kuosmanen 2016), whereby a thorough examination and assessment of major security vulnerabilities was developed. While some are more relevant to Web-Based applications the paper displayed that there are a multitude of techniques such as cross-site and man-in-the-middle attacks, authentication failures, input validation failures and injection attacks which can be harnessed by bad actors upon poorly managed systems.

Kuosmanen critically states that 'Currently the state of security of WebSocket services is stable, meaning no new types of vulnerabilities have been found lately' (Kuosmanen 2016), this helps to imply that if measures are taken against the known security elements a secure protocol can be accomplished. These elements are employed in various scripting styles dependent on the selected language and library, however by following such literature efforts, steps can be taken to develop a robust distributed communication network.

2.3 Existing State of Technology

Distributed industrial control systems require a number of combined elements to operate collectively, allowing safe and reliable function over their life-cycle. Often these systems are a carefully selected combination of separate devices, each meeting strict quality metrics to ensure the overall system can operate to the required level. The major component sections usually consist of:

- Field inputs, such as sensor instruments and discrete devices.
- Controllers, which process the inputs against programmed outcomes, and output the resulting signals.
- Actuators and field effectors, controlled by the output signals.

They also require a communication topology, often with either industrial ethernet or fibre, various protocols and potential links to external systems such as corporate layers or supervisory control and data acquisition (SCADA) systems (Leung 2013). Such overall process or manufacturing systems need to offer reliable performance in both operating system and alarm and status monitoring, to ensure if the reliability diminishes an operator or system supervisor can detect such faults and failures. Legacy approaches have an assortment of protocols, which can require protocol conversion plus interfacing hardware and software for end-to-end data transfer and monitoring (Wang, Luo, Jiang, Xu & Li 2022).

These systems often employ Human-Machine Interfaces (HMIs) for local control and monitoring which can also require a separate programming and software suite, further comms links to be established, and data mapping and replication. As expected, the addition of multiple layers of integrated hardware adds higher complexity and operating requirements to such systems (Wang et al. 2022), further to this, the software options and selections for the distributed controller elements are diverse.

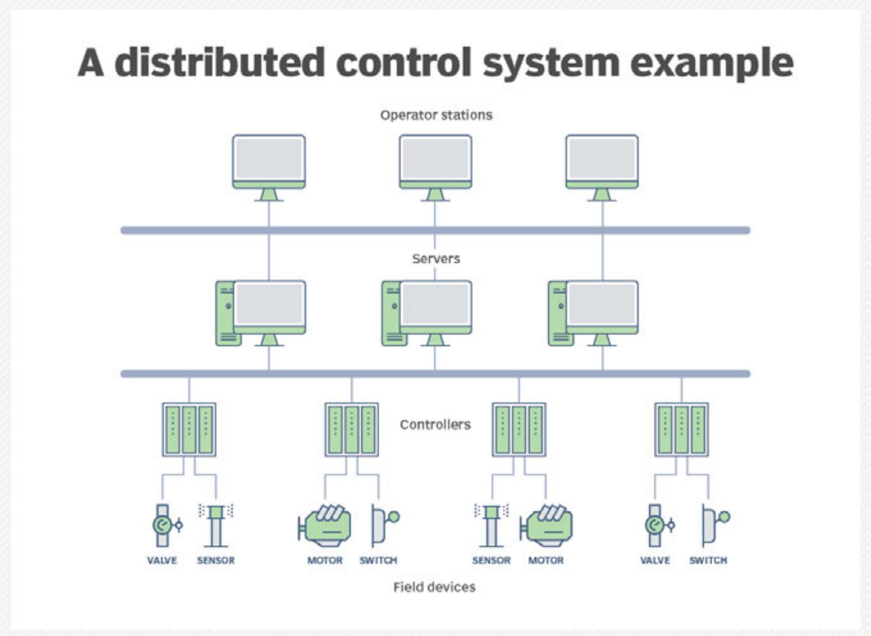


Figure 2.6: A simplified Distributed Control System Architecture example (Gillis 2023).

2.3.1 PLC's and the IEC61131-3 Standard

There currently exists a plethora of hardware and software options to develop control solutions as part of a Distributed Control Architecture, however for such small to mid-sized industrial based systems, PLCs are broadly selected for this particular demand. The original model being a Modicon 084, short for Modular Digital Controller, being released in 1968 (Peterson 2022).

These systems were developed originally from much lower-level logical and sequential control, to allow heavily compacted interfacing with otherwise relay-based hardware control to equipment. Most of the relay hardware in this instance being replaced by electronics and transistor logic, offering improvements in physical size and reliability among other benefits.

PLCs were a critical part of the 3rd industrial revolution but in this regard PLC software development was a very specific industry area limited to proprietary and primitive software development tools (Bonfe', Fantuzzi & Poretti 2001). While the software has progressed with industry needs for the most part, Industry 4.0 and the modern world is demanding greater options and ability to engage with data for business needs.

When utilising these PLC based technologies, the overall structure of the programming approach is predominantly modelled on the standard IEC61131-3 programming languages. This standard is based on five programming approaches, three graphical and two text-based options. Each of these offering certain pros and cons, however the user is always limited in the ways in which they can both manipulate and utilise the control data. The five IEC61131-3 languages which have been formalised since 1993 are listed briefly below (Bonfe' et al. 2001).

1. Ladder Logic Diagrams (LD, graphical). The most common PLC programming form with ladder like visual rungs for logical operations using standard combinational logic and additional assorted features.
2. Function Block Diagrams (FBD, graphical). A highly graphical, representational form for describing signal and data connections between functions and function blocks.
3. Sequential Function Chart (SFC, graphical). A state-transition flow diagram which graphically shows the sequential behaviour of multi-faceted systems. Not considered a full language due to additional requirements which include other languages for some sections of actions and transition function.
4. Structured Text (ST, text-based). A high-level language more aligned to standard, legacy software engineering languages such as PASCAL and BASIC.
5. Instruction List (IL, text-based). A low-level language resembling the assembly type instruction-based programming method.

While each of these approaches requires differing experience and development methods from the user, the final program operation in most cases is heavily structured for certain process models and the ability to create flexible, user-oriented data structures is prohibited.

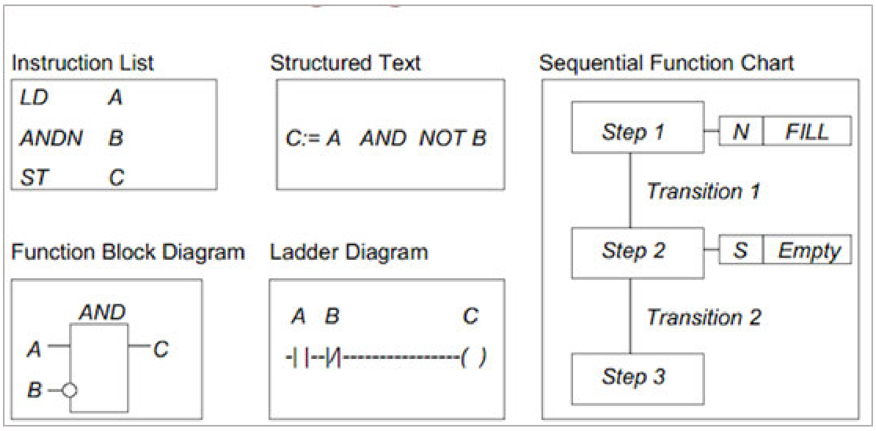


Figure 2.7: Representation of the 5 recognised IEC61131-3 language approaches (Budimir 2018).

The overall structure is formed of variables, function blocks and function calls which are held within a main program cycle, such that control software is cyclically executed, developed from computationally implemented continuous or discrete control methods. Time-driven execution is formed from the execution of control loops, an inherent aspect of the PLC cyclic scan mode of operation (Pang, Yan & Vyatkin 2015).

2.3.2 Distributed Systems and the IEC61499 Standard

As software and control strategies evolve, steps have already been taken to create more flexible systems in relation to both PLCs and time-driven systems across distributed models. The development and use of IEC61499, the Distributed Automation Architecture Standard has seen a transfer to event-driven and component-based encapsulation methods to allow more flexible structures and processes within distributed process control systems (Pang & Vyatkin 2007).

This system allows for a more application-based form, reflective of software engineering principles. The devices form a device model, and the overall system takes on a system model which are related through an overall mapping model. The function blocks differ from IEC61131-3 approaches in that they have an interface and implementation, with both event and data inputs and outputs.

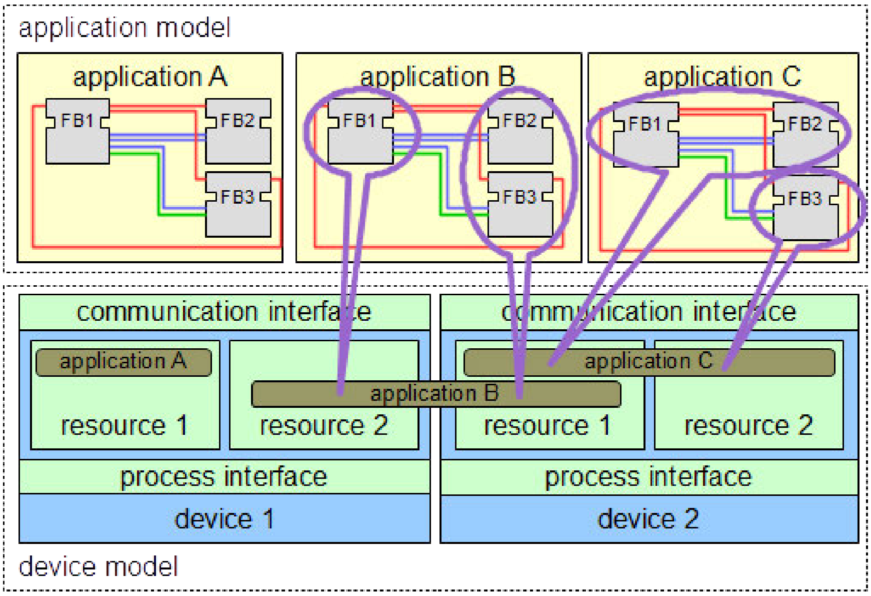


Figure 2.8: System Model overview of the IEC61499 Standard (IEC61499 2024).

2.3.3 Developments and Trends

Research and development has been carried out with the goal of integrating the above listed approaches with some positive results, offering event driven expressiveness coupled with deterministic, time-driven aspects. Key to this is the merging of two distinct modes of operation, the reactive and adaptive nature of event-driven logic, combined with the temporal and synchronous style of time-driven systems (Pang et al. 2015). The combination of these methods allows for flexible, tailored design and implementation, additionally it can assist in the ways that data is both managed and utilised overall.

The addition of event-driven logic holds the ability to immediately respond to state changes, and comparatively, save time and resources in cases where no system change has occurred. Further to the benefits of such trends, IEC61131-3 has also seen more recent use of Object-Oriented aspects such as encapsulation, inheritance, and polymorphism to create more robust, re-useable and modular coding practices (Bonfe' et al. 2001).

Overall, there is a transition occurring whereby modern software philosophies are filtering across to combine with proven legacy methods, originally more hardware oriented and less data-centric in implementation. The two combined, offer benefits which can see control systems perform more efficiently and see essential data become more freely accessible, key traits for modern distributed systems.

2.3.4 Protocols used, Data Mapping and Interfacing Requirements.

Industrial control when distributed in nature can call upon many high-performance protocols to enable reliable, high-speed data-transfer between dispersed controllers and interfaces. It is not uncommon for each network interface to change protocol requiring expensive protocol convertors and separate hardware to be installed in some instances. Industrial Ethernet and Fibre Optic are the most common media for the backbone of distributed topologies. Dependent on the protocol associated, Industrial Ethernet can offer built-in determinism, real-time control with rigid error checking methods while Fibre Optics can add the benefit of long-distance, high-bandwidth data-transfer.

There are two major standards for Industrial Protocols being IEC61158/IEC61784 – these standards define protocols and fieldbus which offer real-time distributed control for industrial systems. Within this framework, Industrial Ethernet encompasses many of these protocols such as Ethernet/IP, Profinet TCP, Modbus TCP and the less common but evolving Real-Time conformant EtherCat and SERCOS III (Knapp & Langill 2015).

The elimination of protocol changes is not possible in most cases, with the majority of such changes occuring at the field interfacing level where PLCs and non-common control components must convert the bi-directional data streams. Due to this there exists a host of available protocol convertor units on the market, each requiring detailed knowledge regarding the process involved to develop data mapping tables and configuration settings to enable such communication. In addition, several of the above listed protocols are often employed with PLCs over the older RS232 & RS485 Standards such as Profinet, Profibus and Modbus, further enlarging the required interface requirements.

Whether a built for purpose microcontroller, industrial PC or PLC is implemented for use, the communication method must be flexible for future needs offering ease of modification, interoperability, reliability, and speed. Information Technology (IT) and Operational Technology (OT) continue to converge, as Industry 4.0 develops unlocking access to edge devices. This convergence aims to allow streamlined access to new data points, previously not seen as essential for business needs due to many factors including lack of available technology and cost barriers (Gundall & Schotten 2021). As can be seen below there is a growing shift towards Ethernet based communications in light of the above factors.

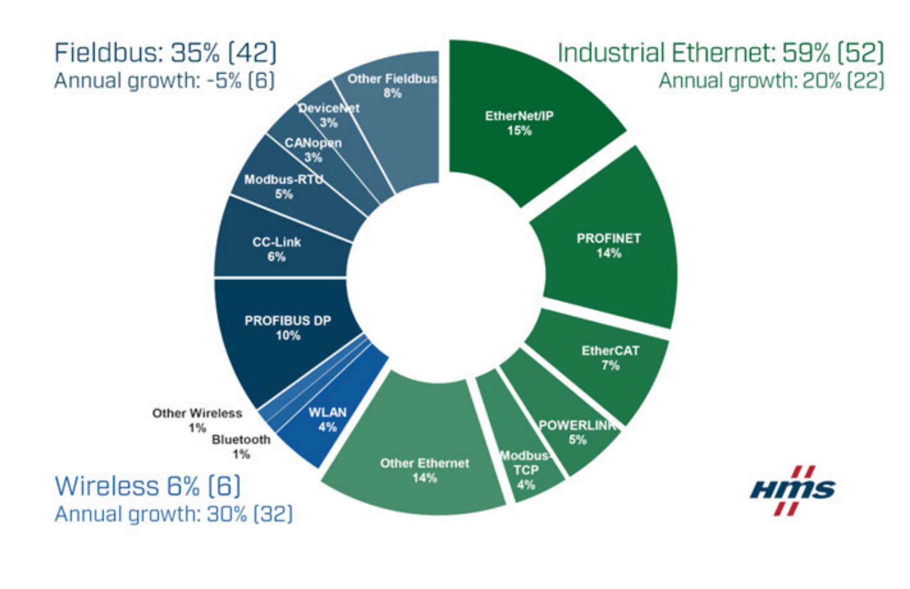


Figure 2.9: Distribution of Industrial Protocols and growth showing ethernet weighting (Fluke 2024).

An important aspect of this project was to explore how an Ethernet/IP hardware layer could be paired with a Websockets protocol layer to allow seamless, real-time capable communications using web-based JavaScript programming methods. This allowed the major control sections to adopt a data communication approach which utilised many of the positive factors discussed above and continued with a project-wide implementation of consistent web-based programming methods, simplifying the use of data and the development of key aspects of the software stack.

2.4 Similar Approaches, Implementations and Benchmarks

Automation and the Internet of Things (IoT) has rapidly developed over the last decade, due to this there exists an extensive resource of research and implementation examples relating to web-based systems, Node.js control approaches, WebSocket communication examples and other highly relevant references. Systems are increasingly providing and consuming more data, giving clearer insight into production cycles and the operation and management of infrastructure, equipment, and smaller sensory devices.

In a study by M. A. Sehr, et al, a review of the emerging Industry 4.0 requirements for industrial controllers was given. Key factors were addressed such as how safety, reliability, security, and efficiency can be achieved using modern programming techniques in conjunction with legacy-based systems (Sehr et al. 2021). As already mentioned, PLCs offer comparatively simple software logic with rugged and robust hardware, being an established factory automation system.

Further within the paper, the authors detail that PLCs lack the flexibility to adapt to more complex automation requirements where integration with smart sensors, wireless and internet-based systems, advanced machine learning and data centric applications may be required. They further mention that the aim is ‘not just to increase flexibility and generality of programming possibilities, but also enforce constraints ensuring predictable, analysable, and reliable behaviour’ (Sehr et al. 2021).

For decentralised control the researchers explore time-stamping event-based logic to assist in deterministic behaviour. They also list catch and handle alarm management techniques for handling exceptions in the running program. It is noted that as the complexity of the system increases, a more disciplined use of memory is crucial to ensure deterministic execution, allowing testability and defining a single specific response for a given set of input conditions (Sehr et al. 2021).

One area which was focused on in this project was developing equivalent flexible structures and abstractions for normally restrictive PLC based functions. An example of similar efforts can be seen in a paper where the researchers attempt to build an abstraction layer for PLC programming using the Object-Oriented features available within IEC61131-3 (Racchetti et al. 2015). In this paper the authors investigated differing approaches to development firstly including a Model-Driven Approach (MDE). This is less favoured due to ease of use but reduced flexibility to personalise designs. Secondly, a Component-Driven Software Engineering (CDSE) approach was explored. This second method emphasised the ‘separation of concerns’, with the aim of creating independent components which can be reused to encapsulate data or functions in a program.

The authors explained the difficulties in design, development and maintenance of such components. They also noted the complex nature of the process required to provide functionality and flexibility, coupled with the importance of well thought out design and organisation of the system. Within their review of available strategies, they found that a design pattern for encompassing a range of select field elements with abstracted control had promising characteristics.

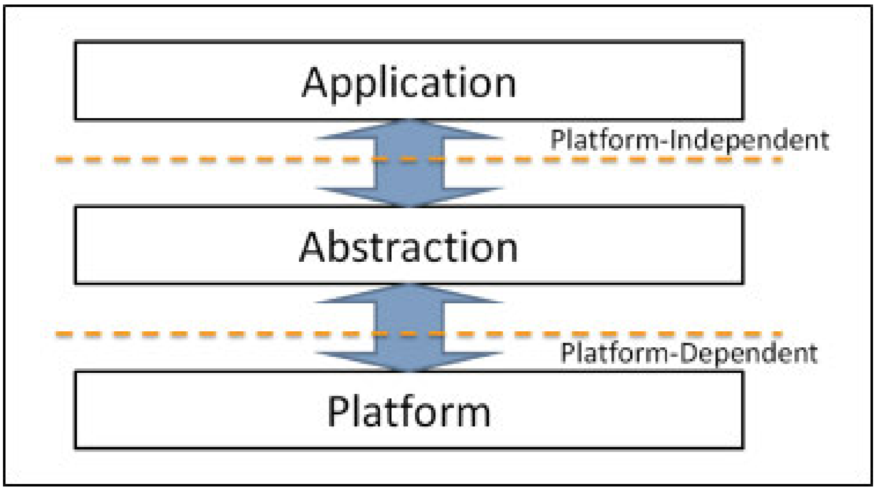


Figure 2.10: Abstraction layer overview common to software engineering (Racchetti et al. 2015).

The term 'generalised device' (Racchetti et al. 2015) is used to describe the object-oriented software component which the researchers developed to encompass a range of simpler discrete devices. However, the software object is limited in scope as they stated, 'it can not model devices with more complex feedback' (Racchetti et al. 2015), referring to motion control in this case.

Two of the key goals in software engineering are modularity and re-useability, these are usually achieved through layers of abstraction and careful definition of classes, objects, and functions to provide the intended operation. In this case the authors provided an Object-Oriented Architectural Design Pattern designed to be a re-useable solution which could hide the implementation details of certain functionality. This project aimed to abide by these principles when establishing system functionality to ensure efficient design and operation was maintained.

Another paper which reviewed key aspects relevant to this project was "A network of Automatic Control Web-Based Laboratories," (Vargas et al. 2011) in which the researchers developed a web connected system for a remote student learning lab. In conjunction with the design element, the team also investigated existing remote systems which included a real-time control loop tied to a central server which asynchronously communicated with a client interface. The researchers explored how to achieve this using Transmission Control Protocol / User Datagram Protocol (TCP/UDP), which exchanged data and commands with a design pattern termed as a "command-based architecture" (Vargas et al. 2011).

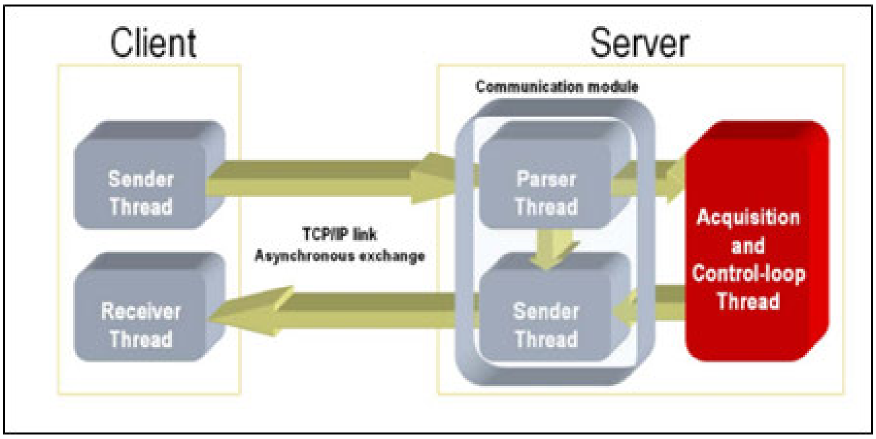


Figure 2.11: Command-based architecture (Vargas et al. 2011).

One sample architecture operated with a remote client sender thread which connected over a TCP/IP asynchronous communication link to the central server running a command parser thread. This software parsing routine was responsible for syntactic interpretation of the incoming requests and execution of the related actions. Inversely, within the server was a sender thread which is tasked with transmitting measurement data extracted from the control loop algorithm, back to the client when directed by a command.

The overall style of this system was that of a client-server model, this had noted limitations when relating to closed-loop control which the authors define. They explained that this model required two separate information loops, a real-time control loop running on the server side and the information & communication loop running asynchronously over the network (Vargas et al. 2011).

When dealing with control system algorithms it is essential to ensure that any time-based systems are correctly managed and separated from the independent, asynchronous communications and operations. This can be achieved by careful segregation and development of modular sections of the overall application. This was another major consideration for this project during implementation.

In terms of separation and abstraction, a research paper by Muhammad Umer et al., looked at a novel approach for creating interconnected, smart power tools for the company Scania. The aim being to create a standardised method of data collection for such systems, allowing enhanced flexibility and accessibility of data via web services (Umer et al. 2018). The authors proposed a system which decoupled existing components and created a Service-Oriented Architecture (SOA).

This consisted of small application components which were 'wrapped', a concept of isolating and hiding the internal functionality of a component, by only exposing the necessary software interface. This approach allowed an interoperable, modular framework for developing solutions with re-useable software (Umer et al. 2018).

This form of architecture permitted the components to interact with each other via data over a network while becoming independent entities within the overall application scheme. Such service-oriented architectures have two distinct component sections, the interface, and the functional implementation.

The component interface allows interaction with other entities by only exposing the functionality of the service used, creating a standardised method which becomes modular and much simpler to manage as the complex internal functions are not exposed (Umer et al. 2018). This allows a scalable, structured system with simplified interactions between intended modular sections.

The authors further enhanced the given system by utilising event-driven architecture (EDA) methods explaining the potential benefits of such a system. It was shown that such a system could generate an event such as an automated message to an operator when a piece of tooling required changing. The use of EDA allows intelligent decisions by utilising event information, compounding this, it was shown that such decisions can be made not only by personnel but by the systems themselves when developed correctly (Umer et al. 2018).

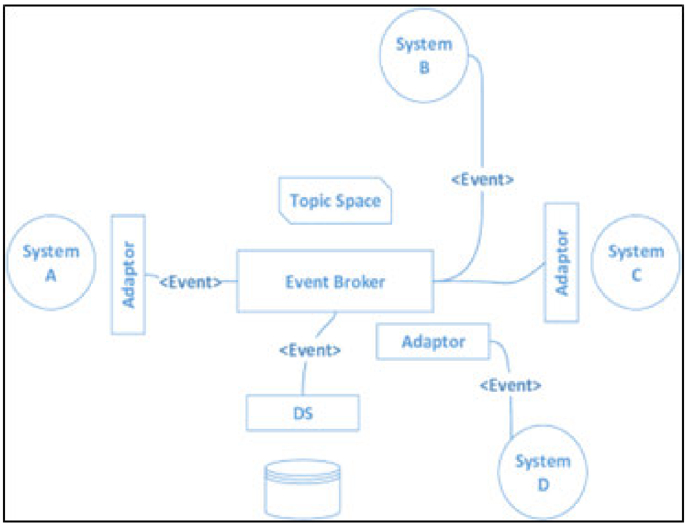


Figure 2.12: Event-Driven Architecture example with standard central broker (Shabani 2018).

A simplified explanation of an event-driven process was also provided within the paper explaining the loose coupling methodology (Umer et al. 2018):

> ”*Only the creator of an event knows that the event has occurred, . . . on the other hand the subscriber to the event only knows about the event and not any details about the creator of the event or the location of the event, unless the event itself contains the information about the creator and origin*”

As highlighted, Industry 4.0 requires the unlocking of previously inaccessible data stores and integration with software and enterprise solutions to ensure companies create a competitive edge and efficient business model. Bellini et al. (2022) investigated these points by implementing a hybrid Distributed Control System (DCS) / SCADA system, which combined an Internet-of-Things (IoT) system with an event-driven distributed data scheme. The hierarchical structure they presented in theory addresses the below elements:

(i) Able to collect data from several sources at different rates.

(ii) Historise high-level data and decisions taken.

(iii) Monitoring the system and generating higher level alarms and actions.

(iv) Graphically representing the higher-level status of the system.

(v) Allowing the connection of production and maintenance activities.

(Bellini et al. 2022).

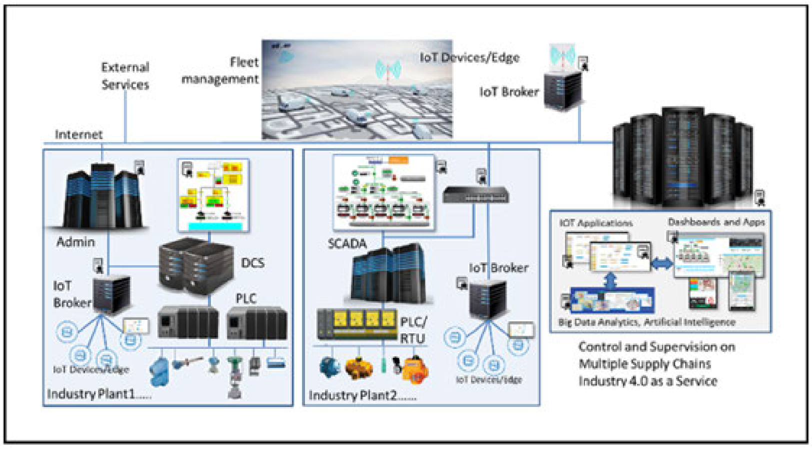


Figure 2.13: Overview of an integration of Industry 4.0 systems (Bellini et al. 2022).

To implement their solution, they called upon pre-built solutions in the Snap4City IoT package and Node-RED software integration system. Snap4City is an IoT development environment and framework useful for implementing supervisory and control aspects of combined production chains. It is open-source and able to provide dashboards of visualised data, microservices and connect sources using protocols including WebSockets, noted for its secure connection capability.

Node-RED is used predominantly as a graphical-based design and integration software for rapid small-scale IoT connectivity and configuration (Bellini et al. 2022). As the name suggests it is built upon Node.js and is suitable for rapid prototyping and connecting different protocols and sources by creating IoT gateways. Due to the graphical nature, the researchers state that Node-RED may not be very effective in performance terms, due to extended round-trip time based on the reading, computing, and acting cycles within the control scheme, coupled with the higher-level implementation of the overall system (Bellini et al. 2022).

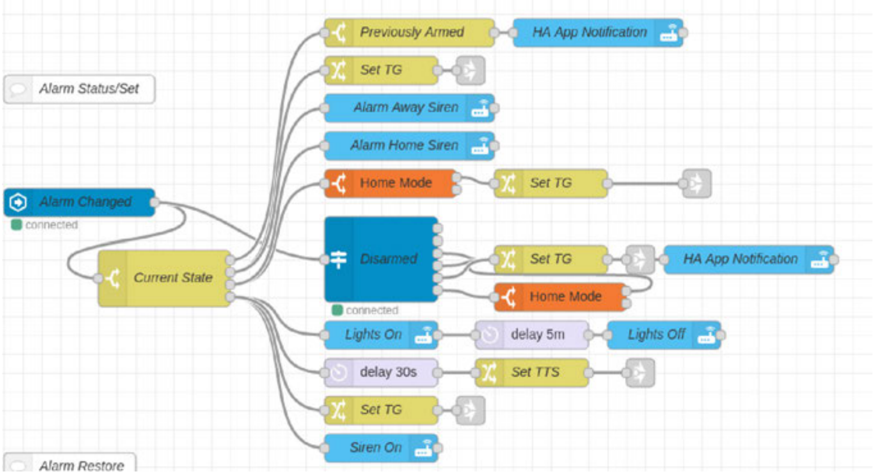


Figure 2.14: Example of a Node-RED workflow into an IoT app (Musings. 2020).

As a system, Node-RED is highly beneficial for IoT use, however it can be somewhat prohibitive for scaling, with the added difficulty in code-review due to the easy-to-use, but hard to verify graphical nature when using on larger-scale, complex designs. This is visible from the above Figure 2.14, whereby many linked interconnections can make complex designs harder to navigate. However, when used for interfacing, the Node-RED system is efficient and greatly simplifies some common tasks by providing ready made modules to the designer.

The paper demonstrated that with the selected technologies a system could be implemented which addresses many key criteria including exploiting big data storage, powerful dashboard creation and executing data analytics to combine with business intelligence. These are quite specific functional criteria, whereby some of the non-functional requirements were also relevant to this project, being:

(i) Robustness (Fault-tolerance and availability).

(ii) Scalability (Small to large system requirements).

(iii) Security (Authentication, secure connections etc).

(iv) Privacy (Compliance, data privacy) .

(v) Openness (Providing possibility of additional modules and functionality).

(Bellini et al. 2022).

An important step in developing industrial technology is defining the benchmarks and key attributes of future system frameworks. Such a framework was developed and explored in the paper "Assessment of Industrial Internet Platform Application in Manufacturing Enterprises" (Li et al. 2021) where the authors listed the major and sub-level indicators for such applications.

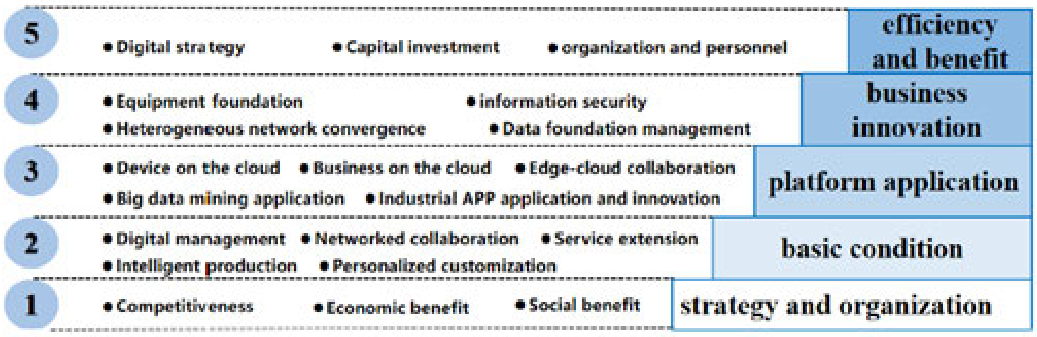


Figure 2.15: Developed framework for assessing an Industrial Internet Platform (Li et al. 2021).

These indicators above reflected many of the major goals and considerations for future systems such as: refined control strategies, accessible and highly utilised data to drive efficiency, optimisation and customisation. While this list is not exhaustive it mirrors the general trends for industrial / enterprise fusion in the modern environment. Many of these themes are reoccurring in business, production and manufacturing environments, this is why they need to be factored into the design of software solutions which aim to merge data sources and intelligent systems.

2.5 Chapter Summary

The concepts that have been examined above have been continually assessed within the scope of this project to ensure that where possible, the software design and implementation satisfied such elements. Regarding the above strategies and results, this project aimed where possible, to incorporate the below points.

(i) Object-Oriented design

(ii) Combined Event-Driven and Temporal Control philosophies

(iii) Error-Checking methods

(iv) Real-Time capable communication networks

(v) Accessible data sources

The aim was to determine if a flexible software application could be implemented as a framework for a Distributed Industrial Control System to encompass many of these Industry 4.0 relevant concepts.

Chapter 3

Methodology

3.1 Chapter Overview

This chapter defines the overall methodology for developing and assessing the implementation of key aspects of a Distributed, Industrial Control System using Web-Based technologies. The chapter addresses the higher-level themes explored in the previous chapter and provides a detailed outline of how selected hardware and software approaches were used to develop solutions. Further, it lists the performance attributes measured during the design and implementation stage of this project.

3.2 Introduction

Distributed Industrial Control Systems involve the merging of robust hardware in combination with carefully constructed software programs and systems, ensuring continuous, reliable operation. This project did not aim to replicate expensive, well-tested, industry-standard hardware, available off the shelf for use. Instead, it aimed to provide a reliable system, capable of implementing software processes and communication which could demonstrate novel and refined approaches, delivering an easy to model and modify, Distributed Control System approach.

To achieve this chosen approach, a selection of embedded hardware was utilised in conjunction with some general network equipment allowing a base topology to build above. Further to this, interfacing hardware was utilised where possible on the embedded systems to allow connection, control, and sensing of field sensors and actuators, predominantly to explore potential options. The major focus of this project was to develop software solutions to satisfy common legacy and modern control system requirements. Therefore addressing event-driven and temporal-sequential operations, communication protocol implementations and other functional aspects of the control system such as database operations and visual applications.

As with any major research project it was important to address and correctly identify all risks associated within the scope of works. The inclusion of a detailed Risk Management Plan was therefore provided to control the hazardous aspects of this undertaking. A project timeline was also included which highlights the major deadlines and sections to ensure the project met required timelines for completion.

3.3 Project Documentation

It was anticipated that the initial implementation of the base topology alone would contain challenges including compatibility issues, configuration nuances and requirements to constantly refine the approach. Further to this, sourcing of suitable equipment for use and developing electronic solutions could be required for interfacing systems of different supplies and varying power and sensing requirements.

It was essential given the probability for setup issues that a clearly documented approach was used for all system development, both hardware and software. It was also a key criteria for producing a high-quality research project and meeting the capability of demonstrating the targeted outcomes. Documentation for the given solutions comprised of diagrams, photographs, flow charts, tabulated results and software records including version control where necessary. This process was important to ensure that any efforts to follow the setup and development could be easily followed and replicated by others.

3.4 Risk Management

The Risk Assessment completed for this project can be reviewed as Appendix B at the end of this document.

3.5 Project Breakdown and Timeline

This project was broken into a number of stages and sections to research and develop solutions within the defined scope. By segregating sections a progressive approach was maintained which clearly developed the key points and documented the project steps taken to produce the final results.

Thes major sections include:

- Literature Review - Initial review and research of existing technology and approaches.
- Methodology - Reviewing Hardware and Software approaches and the intended implementations and solutions for creating the project.
- Design - The main section whereby the project was developed and tested to determine suitability and capability of Web-Based technologies for distributed control.
- Results - Review and collation of the results from project development and testing.

The following page shows the associated project timeline used to manage deadlines and project progress over the course of this research undertaking.

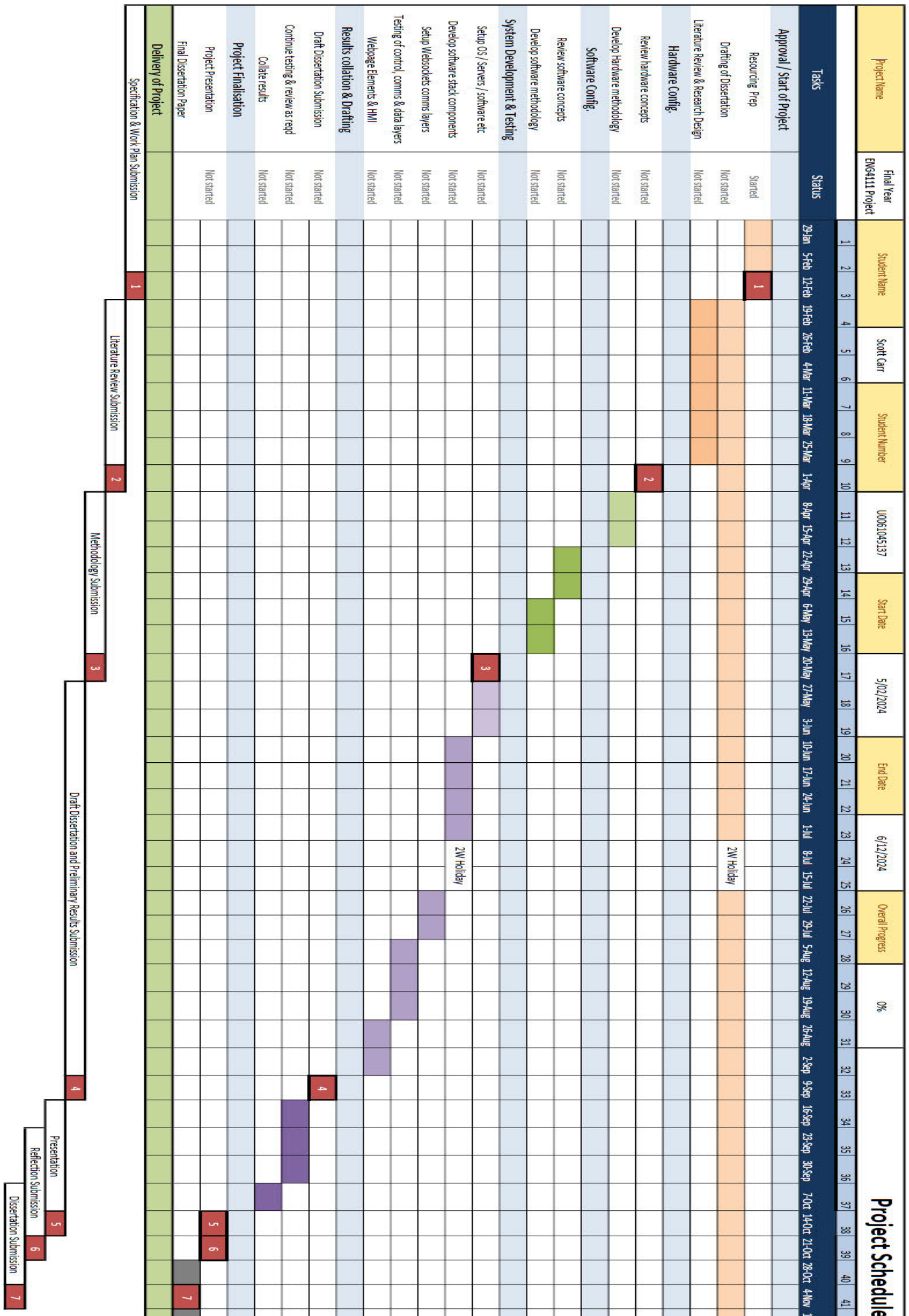


Figure 3.1: Timeline of critical events as part of the research project.

3.6 Hardware

There were a number of hardware solutions available for selection and use within this project, however due to the availability, low-cost and flexibility of the Raspberry Pi brand of embedded boards, these were ultimately selected for use within this scope. Further justification is given for this selection, accompanied with the associated details of the hardware selections within this overall section.

3.6.1 Distributed Control System Architecture

To develop a model distributed architecture a three board system was been chosen as it allowed multiple communication channels to investigate various protocol and control strategies.

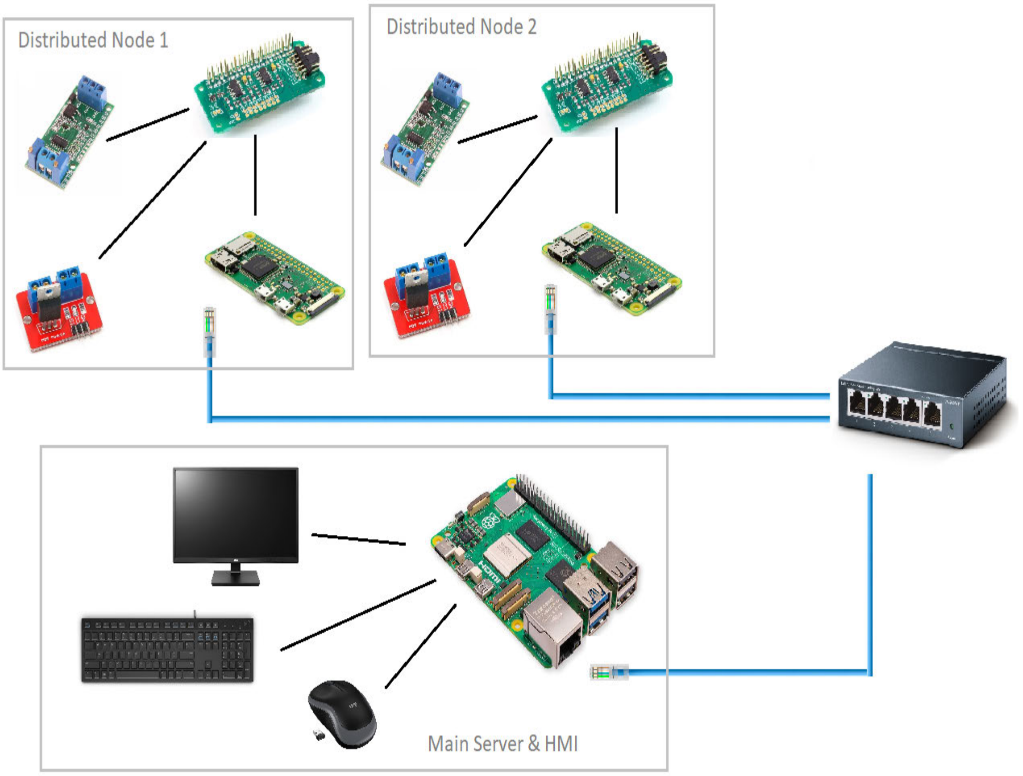


Figure 3.2: Overview of the Distributed Control System Architecture / Topology .

To allow this system to operate, a more capable central server, embedded board was selected which could perform database operations, act as the Human-Machine Interface (HMI), and handle more advanced computational functions. The board selected for this role was the Raspberry Pi 5.

The two other separate embedded boards were the smaller form, reduced capability units, which are used for basic control, data acquisition and interfacing to field elements. For these boards the Raspberry Pi Zero W was selected for each.

The three units communicated over an Ethernet layer 2 data link, via an interconnected network switch. This physical layer allowed the WebSockets protocol to be established and communicate between the three running systems.

While hardware is a major aspect of Industrial, Distributed Control Systems, this project intended predominantly to focus on how software elements could be implemented to create solutions on top of standard embedded hardware. A challenge of this was to source embedded solutions which allowed the implementation of a smart, distributed architecture, capable of developing software within the given scope whilst minimising the overall budget. One major benefit of the Raspberry Pi models was that there existed a range of different models allowing for flexible solutions which could be integrated together, shown in the design of this project.

The following sub-sections detail the various hardware selections which were sourced and utilised for this project. Each sub-section gives appropriate justification, listed with the inherent features.

3.6.2 Raspberry Pi 5

The Raspberry Pi 5 was a high-performing embedded board, more than capable of providing the hardware and software performance requirements of this project for the main central server.

The onboard processor was a Broadcom BCM2712 2.4Ghz quad-core 64-bit Arm Cortex-A76 CPU (Raspberry-Pi. 2023). It featured 512KB per-core L2 caches and a shared 2MB shared L3 cache.

Figure 3.3: Raspberry Pi 5 Embedded Board (Raspberry-Pi. 2023) .

There were three other alternate boards investigated for the role of central server. These were the BeagleBone, Jetson Nano and Arduino Uno. The Arduino was quickly eliminated as an option due to the reduced capability to handle visual output and independent operating systems. The Jetson Nano was a more powerful unit with more functionality, higher processing capability and interfacing options however the cost was prohibitive. The BeagleBone was the closest competitor and while only slightly more expensive it was not chosen, as a complete Raspberry Pi solution was decided, due to the availability and low cost of the accompanying Pi Zero W boards. A list of specifications is provided in Appendix D.

3.6.3 Raspberry Pi Zero W

The Raspberry Pi Zero W was chosen as the embedded board for the distributed nodes for a number of reasons. The total cost per unit of $24.50 +postage was far less than most alternatives. The units had a small form factor, had many libraries available for use through npm and importantly had Ethernet capability available when used in conjunction with an adapter to support a WebSockets implementation.

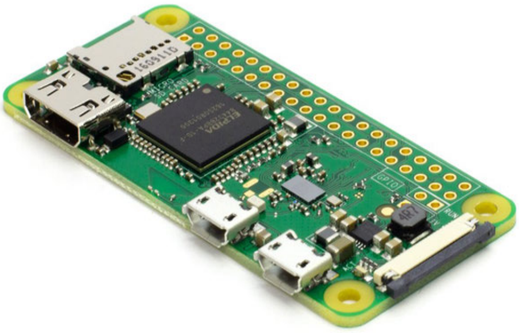


Figure 3.4: Raspberry Pi Zero W Embedded Board (Core-Electronics. 2024) .

While the units were low-cost and small they still had graphics capabilities and compatibility with required software such as NodeJS. The units could also be run via Secure Shell (SSH) to allow a secure, remote connection for programming and file transfer using the popular open-source terminal emulator, Putty. A list of specifications is provided in Appendix D.

3.6.4 Raspberry Pi ADC Module

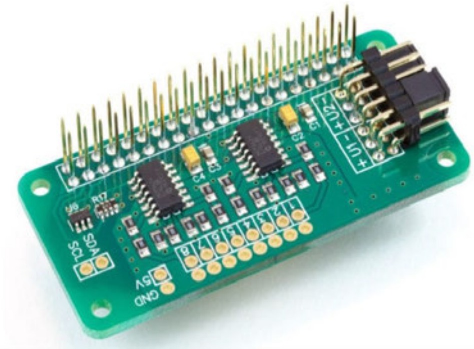


Figure 3.5: Raspberry Pi ADC Module (PiHut. 2024) .

The Raspberry Pi ADC was an analog to digital converter board which could be placed on top of the Pi Zero board, enabling power through the GPIO pins and adding capability for analog signal processing. The board achieves this by using two on-board MCP3424 Analog to Digital converter chips. These chips are addressed and work over I2C communication to provide 8 analogue inputs for use. The addressable range for each board is 8 allowing up to four boards to be stacked providing 32 inputs per local controller. There is also a 3.3v to 5v logic converter to allow easy interfacing with the lower voltage Pi supply (PiHut. 2024).

The MCP3424 included an integral 2.048V reference voltage with a full scale range of 4.096V. There was also a programmable gain amplifier with ranges of x1 to x8 prior to the ADC conversion. Using the I2C interface the data rate and resolution could be adjusted from 3.75 samples per second (SPS) for 17 bit up to 240 SPS for 11 bit resolution. (PiHut. 2024).

3.6.5 PWM 3.3V/5V Voltage Converter Board

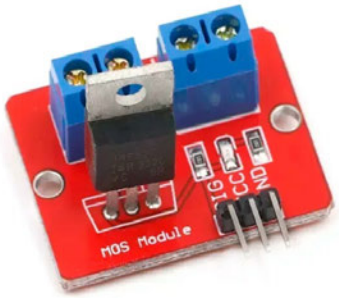


Figure 3.6: PWM 3.3V/5V Voltage Converter Board (Amazon. 2024) .

Another addition to the hardware interfacing to increase the available voltage driven from the Raspberry Pi Zero was a 3.3V to 5V Pulse Width Modulation (PWM) driven converter board. This board boosts the 3.3V output from the GPIO pins and the ADC board to a more flexible 5V supply where required.

3.6.6 Ethernet Unmanaged Switch

To allow connection of the three separate embedded systems over a network, an unmanaged Ethernet switch was selected offering a simple plug and play physical network to be enabled. The unit was powered from a separate 24v DC supply and could support up to 5 connections. This allowed implementation of the WebSockets Protocol over the Data Link Layer.

Figure 3.7: TP Link 5-Port Ethernet Switch (TP-Link 2024).

3.6.7 24 Volt DC Power Supply Unit

Due to the requirement for a number of working voltages within the project the decision was made to select and purchase a 24 Volt Direct Current (DC) power supply unit. This unit had a 3A 72W rating providing sufficient power for all necessary devices within the project.

Figure 3.8: 24 Volt DC Power Supply (ebay. 2024*b*) .

3.6.8 Miscellaneous Equipment

There was a number of smaller items required to allow connections to be established and interfacing and display to be achieved as part of this project. These items are listed within Sub-Section 4.2.3.

3.7 Software

3.7.1 Raspian and Ubuntu/Linux OS

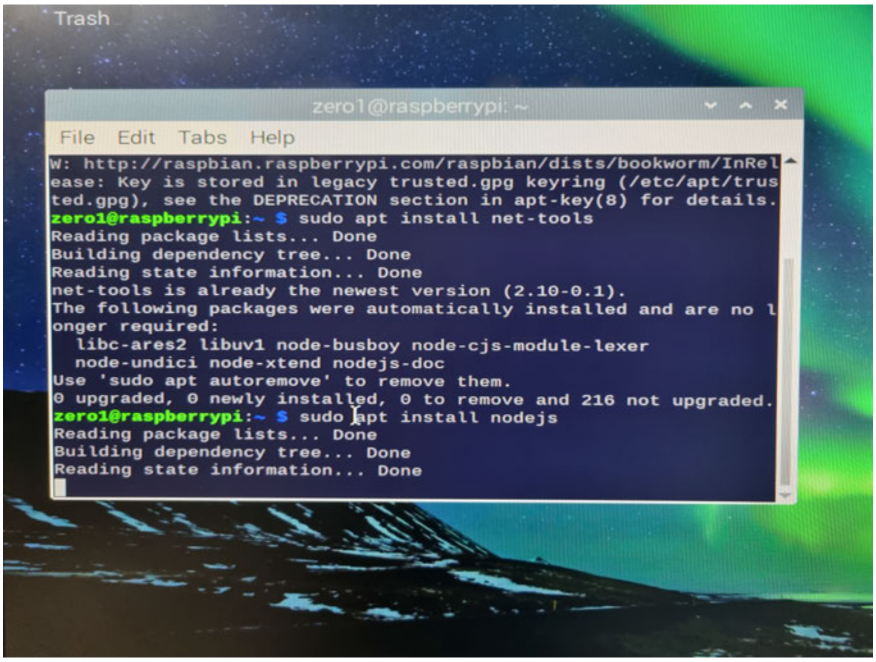
There were two separate operating systems (OS) selected for the three separate Pi boards. The original intention was to utilise the Linux based Ubuntu 20.04.1 LTS distribution for the Pi Zero W boards. This choice offered some added package management system software and program options, however it was discovered during some basic initial setup that the Pi Zero W was not completely supported after initialisation issues and some further documentation review. As a result the Raspian Bookworm OS was selected and successfully installed and run on the Pi Zero W embedded systems.

It was still preferred to progress with the Ubuntu install for the Pi 5, therefore Ubuntu 23.10 was used for the Pi 5 operating system. This was successfully loaded on the Pi 5, offering more advanced package services, an easy to use interface and some more familiar methods.

Both operating systems importantly offered terminal access for downloads, aiding in rapid installation, configuration and setup. They could also be used with NodeJS and the associated libraries. It was ensured that the two versions were also established and proven stable, to add to the reliability, as the latest versions can in cases still hold performance bugs and software issues.

3.7.2 NodeJS & NPM

With the availability of terminal installs on both operating systems NodeJS and npm could both be installed in a reasonably easy process. To install, the user can open a terminal instance and : “sudo apt-install node”. This sets the user as super user similar to administrator in windows based environments, allowing root privileges for download and file system management. The statement also calls the up-to-date software package from the appropriate location, known as a repository in Linux-based systems, to download and install. Both nodeJS and npm were used extensively within the project with each use case documented further within the Functional Design section.

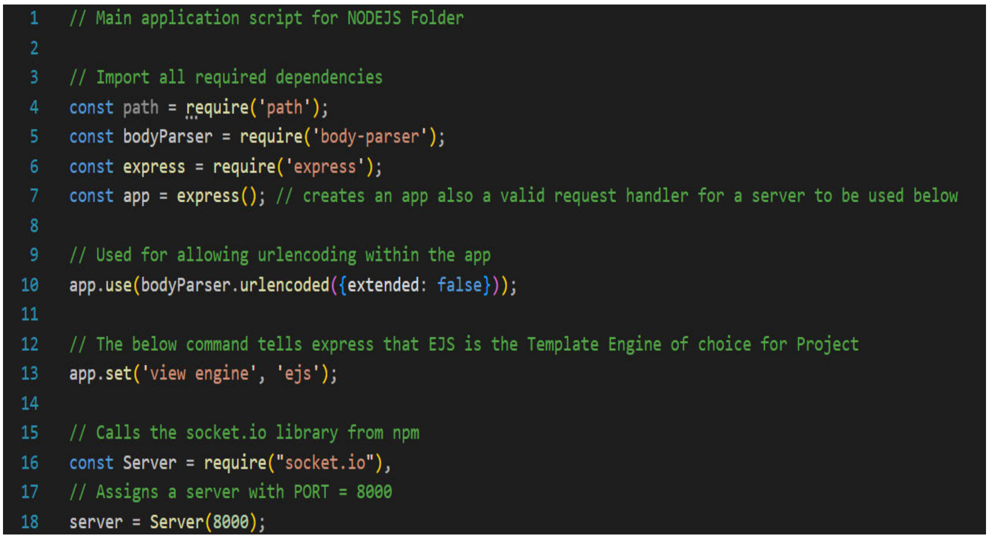


```
W: http://raspbian.raspberrypi.com/raspbian/dists/bookworm/InRel
ease: Key is stored in legacy trusted.gpg keyring (/etc/apt/trus
ted.gpg), see the DEPRECATION section in apt-key(8) for details.
zero1@raspberrypi:~ $ sudo apt install net-tools
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
net-tools is already the newest version (2.10-0.1).
The following packages were automatically installed and are no l
onger required:
  libc-ares2 libuv1 node-busboy node-cjs-module-lexer
  node-undici node-xtend nodejs-doc
Use 'sudo apt autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 216 not upgraded.
zero1@raspberrypi:~ $ sudo apt install nodejs
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
```

Figure 3.9: An open terminal instance with request to download nodeJS .

3.7.3 ExpressJS & EJC

ExpressJS and EJS could also be installed ready for use using the terminal directly. These were used within the nodeJS app, this was achieved by importing into the source code of the application program once installed using the *require* directive within the application dependency section.



```javascript
// Main application script for NODEJS Folder

// Import all required dependencies
const path = require('path');
const bodyParser = require('body-parser');
const express = require('express');
const app = express(); // creates an app also a valid request handler for a server to be used below

// Used for allowing urlencoding within the app
app.use(bodyParser.urlencoded({extended: false}));

// The below command tells express that EJS is the Template Engine of choice for Project
app.set('view engine', 'ejs');

// Calls the socket.io library from npm
const Server = require("socket.io"),
// Assigns a server with PORT = 8000
server = Server(8000);
```

Figure 3.10: Example node application source code with dependency listing .

3.7.4 MySQL

It was intended to explore various database aspects within the project once the lower-level operations were complete. The aim of this was to demonstrate the ability to easily capture data from the end device and map into the local database for further internal and external use. Some of the features to be addressed were database definitions and structures, timestamping capabilities for historisation and standard transaction operations known as create, read, update and delete (CRUD).

MySql was the selected program to be used as after initial investigation it was seen to be highly compatible with Raspberry Pi systems, relatively lightweight in software terms and was the most popular of Pi users, offering a large amount of associated documentation to assist in development.

Where this system was implemented, all steps and processes to install and implement were documented within the project

3.7.5 MS Visual Studio Code

Microsoft Visual Studio Code or VS Code as it is otherwise known was the selected software Integrated Development Environment (IDE) for project tasks. While Geany and Thonny were two IDE's already on the Raspberry Pi Zero W system they were not as easy to operate and have far fewer features. VS Code was installed on the Pi 5 system and offered a relatively low system overhead, however, it was not compatible with the Pi Zero W due to system constraints. In managing this, all coding was completed on the Pi 5 and transferred onto the Zero W units via a shared file system over the Ethernet Local Area Network (LAN). In limited cases code was updated direct using Geany, for minor modifications.

3.7.6 Putty

Putty is an open-source secure shell terminal emulator which can allow remote access to consoles such as a Raspberry Pi Zero. Due to the addition of two extra monitors for the project development, Putty was not needed, however it offers connectivity features for reduced systems like the Pi Zero W which add flexibility to the units. It can be used to remote access the file system to run programs and make changes, this is known as running in 'headless mode'.

3.7.7 Git and GitHub

Git is an open-source version control system software which can be used to manage development of software projects. GitHub is a cloud-based version control platform which can allow the same functions.

In this project, GitHub was used to ensure modification and project development were controlled so that errors and improvements could be documented and managed. With any software development it is important to ensure rollback and recovery can be achieved if a system modification results in errors, this program helped mitigate this issue during the project.

3.8 Details of Proposed Methods and Testing

There were a number of individual system characteristics which were investigated as part of the overall distributed control system to determine if suitable performance or specific operation could be achieved using the assigned Web-Based software systems. The below Figure 3.11 shows the initial intended, high-level structure of the software to be developed and tested for the overall distributed control system.

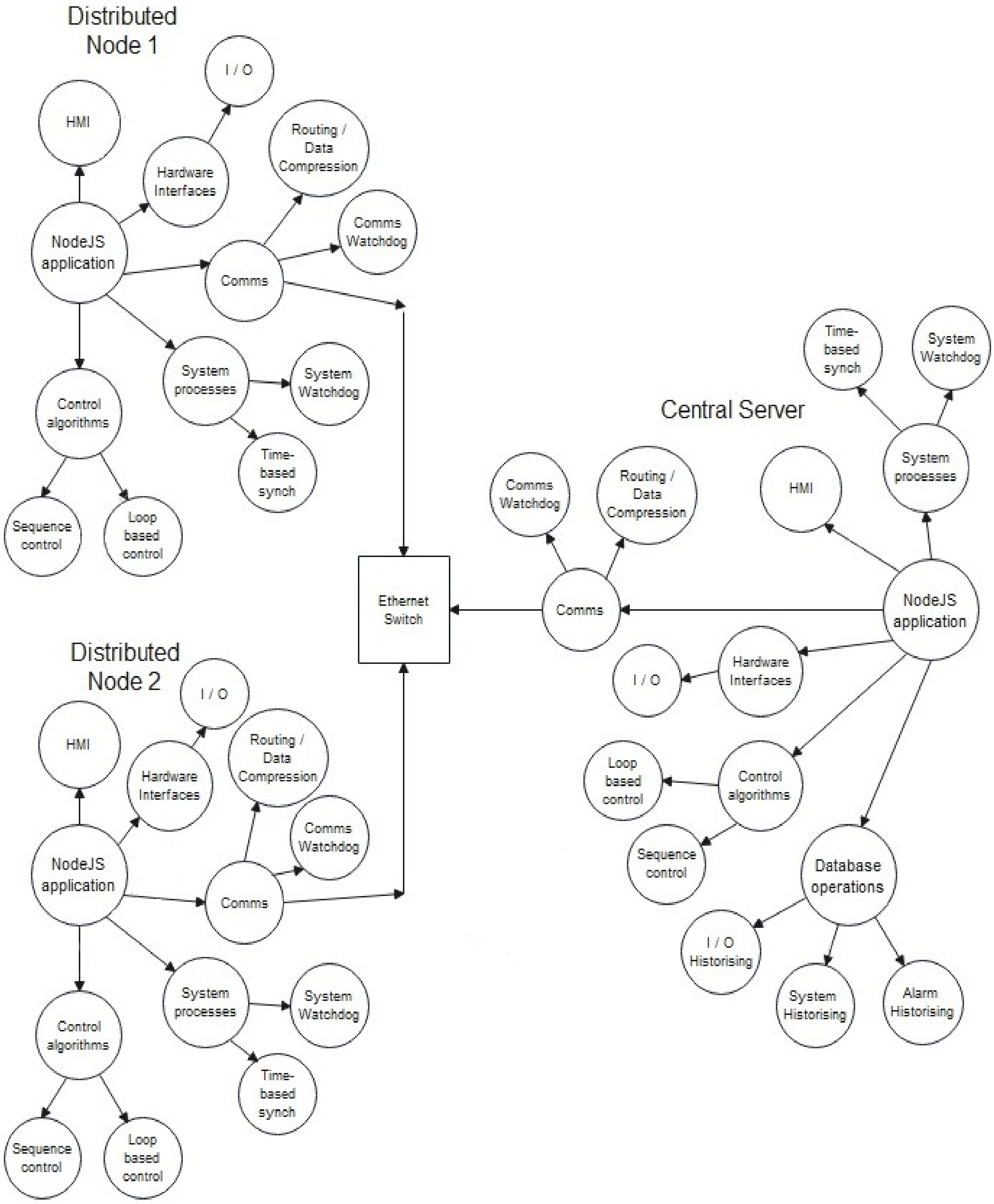


Figure 3.11: The basic intended software topology for the Distributed Control System.

Each of the major components were to implement software routines and object-oriented practices to allow a modular, inter-connected system sub-structure. This reflected some of the core elements of IEC61499, being: Object-Oriented encapsulation of functions and methods, nesting of objects to develop complex interfaces while keeping data wrapped, hardware abstraction and flexibility in system design.

The below project sections are also aligned with the timeline overview provided earlier in the Methodology section.

- Individual Control Server Applications
- Communication Network
- Central Server Scheduling and Timekeeping
- Database setup and operation
- Human-Machine-Interface (HMI)
- Class-Based Modules for control and data handling
- Further Sequential Testing

3.8.1 Individual Control Server Applications

The first step of the project implementation and testing was to setup the three devices with NodeJS applications. This was to be the base system in which all further control programming, communication configuration and interfacing was built from. This step required initial installations, configuration of the required dependencies and any additional libraries in conjunction with separate program development. This allowed a main application to be built which was the interface for all other software sections and sub-systems.

Some rudimentary general purpose input and output (GPIO) functional testing and mapping to the IO interfaces was also attempted and developed within this section. This was initially to indicate the ease at which further interfacing could be achieved to assist with task scheduling and completion.

3.8.2 Communication Network

The next step of testing was to establish the communication setup with the Ethernet Data Link Layer. This initially consisted of physical connection and ping tests to confirm lower level connectivity. Then each distributed system had a WebSockets Server-Client which had specific address ID / data handling, event driven messaging and communications parameters.

Features and measurable characteristics such as Latency, Bandwidth, maximum and minimum data rates, failure detection, watchdog cycles and transmit actions such as broadcast and individual transfers were all to be evaluated within this section. These elements were not completed at the same stage of the project, dependent on project requirements. Further elements were also developed to assist in functionality as required.

3.8.3 Central Server Scheduling and Timekeeping

Once all devices were connected and capable of sending and receiving data the aim was to develop a time-dependent system framework for the overall Distributed Control System operation. This was to start with building a schedule and a sequenced, priority based framework for the system. The exact nature of this scheduling was defined and developed within the design section requiring the separation of cyclical-based tasks from event-driven tasks and appropriate hard-time deadlines which tested whether system priorities and functions can be maintained reliably.

A key role of the central server on the Pi 5 was to create an ordered and structured system to acquire and process all data from the distributed nodes and within its own interfaces. While event-driven execution models are said to be deterministic, giving known responses to set event-triggers, the nature of industrial systems requires that time-based responses can be proven especially regarding safety systems. Therefore the time-based operation of aspects of the central server were seen as critical to determining suitability of such a system in the overall scope of this project.

By using JavaScript *promises* within the NodeJS environment the aim was to ensure system tasks and communications were executed to set time frames. This ensured data updates were achieved within the tolerances of set watchdog timers and allowed the main application to check system parameters at set, re-occurring time intervals.

3.8.4 Database Setup and Operation

This section required the implementation of a database on the Pi 5 which was to connect to the central server to allow persistent data storage and access. It was connected so that data over the WebSockets protocol was transmitted or received by the database via the NodeJS application.

Historian capabilities were investigated for recording and time-stamping Input-Output control data and system parameters. These capabilities would assist an industrial system with managing error checking, data failures and reliability, and recording of events and alarms. Again, this was a critical aspect of the project which demonstrated how such a web-based design could effectively make data both accessible and easily manageable, key traits of modern Industry 4.0 systems.

3.8.5 Human-Machine-Interface (HMI)

There was a section allocated to develop graphical interfaces as part of the overall Distributed Control System. Initially this entailed the set up of simple graphics to display parameters and live data or retrieved data from the database. This was also to be utilised to explore how simple discrete control strategies could be managed across networks using the HMI interface to control or monitor elements of the system. However, due to time restraints this element was not deeply explored with simplified discrete data transfer only completed.

Time permitting more complex graphical elements were to be attempted however it was envisaged that the graphics would predominantly be used to assist with simple testing, interfacing and monitoring functions. The intention was that these functions would provide the assurance that more complex tasks could be built from the fundamental methods used for lower-level operations.

3.8.6 Class-Based Modules for Control and Data Handling

An ongoing task within the scope of the project was to stay aligned with the IEC 61499 principles by building Object-Oriented Structures (OOS) to develop class-based data handling techniques. When referring back to Figure 3.11, many of the high-level components of each application would require specific, purpose-built class modules for dedicated functionality.

A major benefit of following OOS practices was the ability for component re-use and nesting of objects to create flexible functions. Such methods also assisted in data wrapping, when executed correctly this allowed only the exposed methods to interact with data through interfaces. This essentially created a black-box model for data processing, greatly simplifying the readability and operation of code within the application. These approaches were followed where possible to build system components which reflected such benefits.

It was expected that objects and classes were to be defined and used in the below areas as a minimum:

- Database Interfacing
- Hardware and IO Interfacing
- Watchdog operations
- Communication components
- Alarm management modules

3.8.7 Hardware Interfacing

Time permitting the use of more advanced analog Input-Output devices and signals was to be researched and tested. If successfully achieved such functionality would show that accurate, high-resolution signals could be managed within the control scope.

This is why the Pi ADC units were included as part of the hardware specification. This task was to combine with the above stated class and object-based practices to create interfaces, which would simplify how the system could interact and operate with more advanced sensory and control elements.

3.8.8 Further Sequential Testing

Another potential extension of project research and testing was to be development of a simple hypothetical batch or phase-control system which would require further hard sequencing to test the NodeJS system for deterministic operation. This section was dependent on available time resources and the outcomes from initial system sequence testing from work within the Central Server Scheduling and Timekeeping Section. This was not completed.

3.9 Records and Data Analysis

The core goals of this project were to research, develop, test, and assess the suitability of chosen Web-Based software and hardware implementations in relation to the project theme. To assess and determine performance some metrics were needed relating to the above reviewed sections. Some testing was to be qualitative in nature, such as whether certain software methods met the high-level guidelines of IEC 61499 or IEC 61131. Alternatively, some outcomes were to be qualitative, such as the bandwidth, data rates and latency of signals, relating to the communications of the system.

Below is a list of highlighted outcomes / metrics which were to be assessed and recorded during the design stage.

- Recording of defined sequential testing outcomes.
- Latency and bandwidth / loading tests with WebSockets.
- System and NodeJS minimum and maximum execution times.
- Central Processing Unit (CPU) and Random-Access Memory (RAM) usage for each embedded system.
- Analog signal resolution and sample rates achieved.
- Watchdog time settings achieved.
- Records and capture of database timestamp operations.
- Records of communication errors, up-time and issues.
- Overall methods and programs used for testing and recording.

3.10 Chapter Summary

This section reviewed the major component sections for both hardware and software which were to be utilised within the project. It also outlined the intended methods and separate elements which were to be focused on when carrying out the design and testing phase of the project.

From these steps, the design phase was able to determine the suitability of certain individual elements of the Web-Based design, and also aimed to deliver an overall suitability of such a combined system, for the intended industrial application.

Chapter 4

Functional Design

4.1 Chapter Overview

This chapter explores the design approach utilised within the project to create the distributed control system architecture. It initially reviews the hardware design and shows how the components were connected and tested to get the basic system ready for further software development. It then continues to detail the software development cycle including the structure and program flows, dependencies and system components.

4.2 Hardware Design and Construction

The hardware for the developed system was built on Raspberry Pi Micro-Controller Units (MCU's). The designed system had a Pi Five unit as the central server and two individual Pi Zero units as separate clients. Further modifications were completed on the Pi Zero boards to enable analog signals by addition of Pi ADC top-hat boards. There was also a TP-Link 5-Port Gigabit Switch used for enabling the network in conjunction with a NetLink Ethernet Media Convertor.

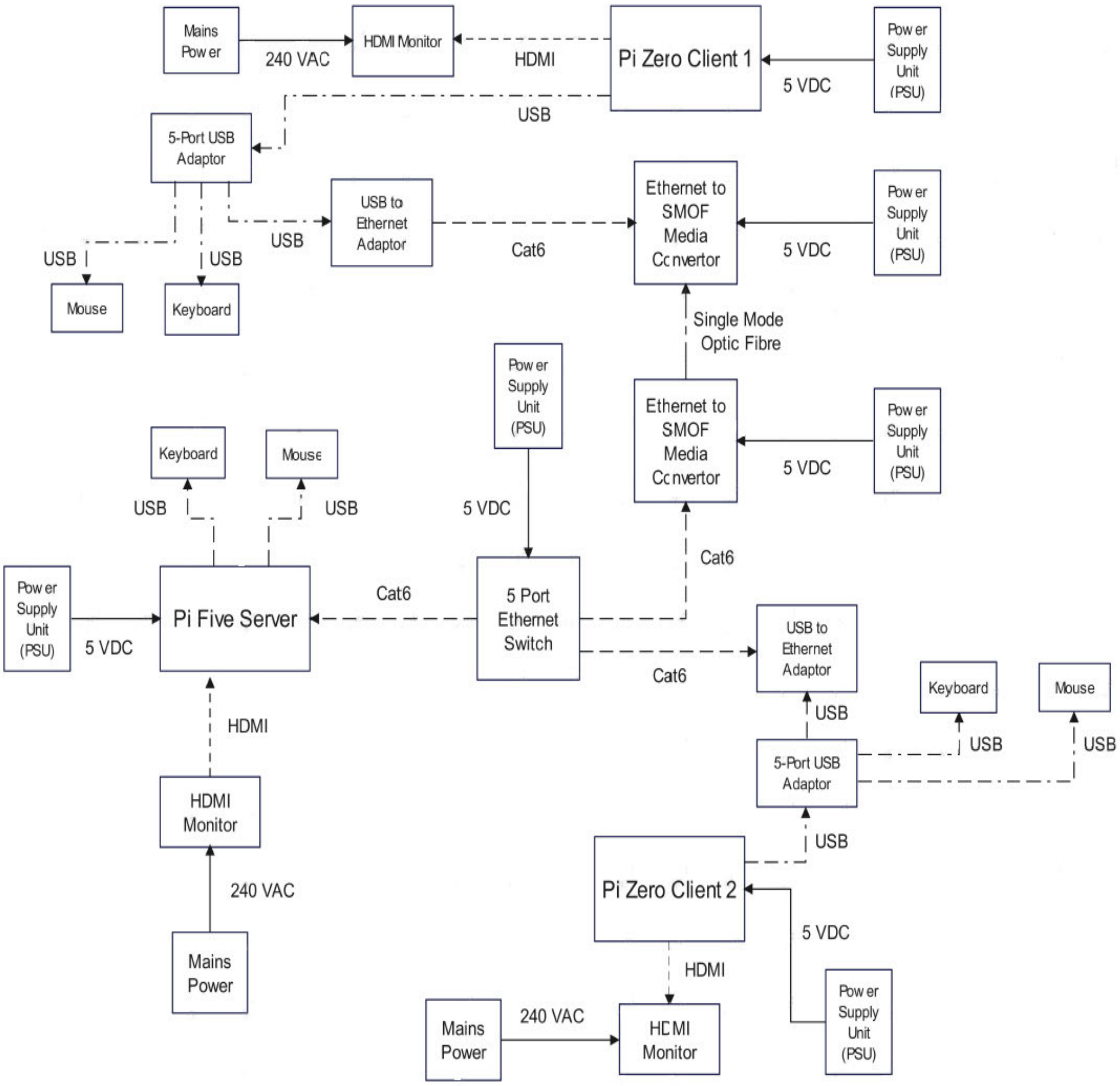


Figure 4.1: Block Diagram of all Power, Communications and accessory connections.

4.2.1 Raspberry Pi Five - Main Server

The Pi Five was a 4GB model and was provided with 4 x direct connect USB points. A Pi Five specific 5.1V 5A power supply unit was purchased which was necessary to allow the unit to run without shutting down. This was discovered initially when an error was shown in the top right of the monitor display, upon energisation with the loaded operating system. The required connections to the unit include:

- 1 x Power Supply Unit (USB-C connection)
- 1 x Micro-HDMI (A Micro to Standard HDMI adaptor was used for monitor connection)
- 1 x Wireless mouse USB connector
- 1 x Keyboard USB connection
- 1 x Cat6 RJ45 Ethernet connection

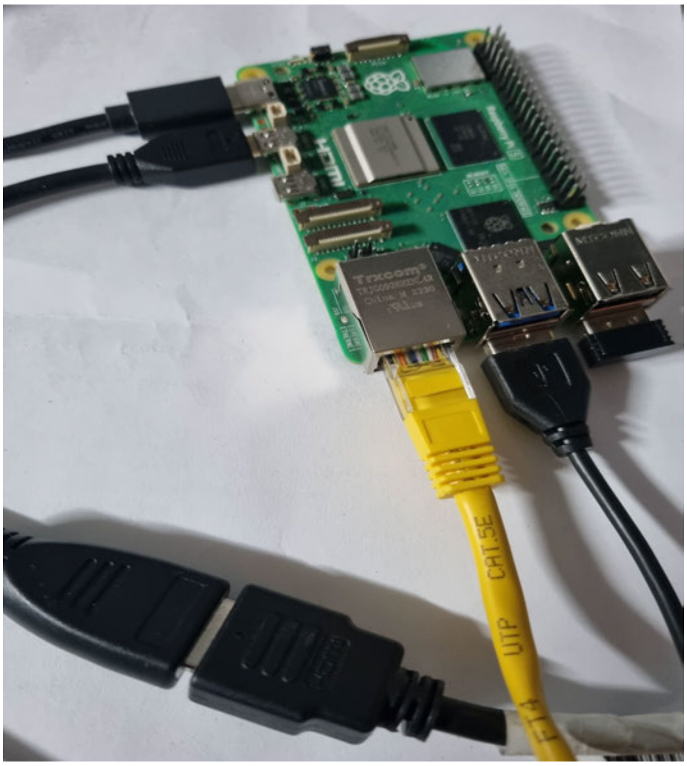


Figure 4.2: Image of the hardware connections to the Pi Five unit.

4.2.2 Raspberry Pi Zero Units - Distributed Clients

The two Pi Zero units while compact and capable, had a limited number of connections and no Ethernet RJ45 connection. As the two Zero units were the distributed clients within the overall system they required input / output (I/O) interfacing. To allow this, there were some preliminary modifications required to the boards. This included the addition of the Pi ADC Top-Hat module which added analog I/O capability to the boards as specified in 3.6.4 of the Methodology section. To achieve this, soldering of the board was required including; the GPIO header pins, 8-pin terminal block, I2C address pins and addition of 4 securing circuit board bolts. Once completed the board was ready for connection and use. The below board connections were initially required:

- 1 x Power Supply Unit (USB-micro connection)
- 1 x Mini-HDMI (This connects to Mini to Standard adaptor for HDMI monitor connection)
- 1 x Micro-USB (Connected to Micro to USB-A adaptor, then to 5-way USB-A Hub)

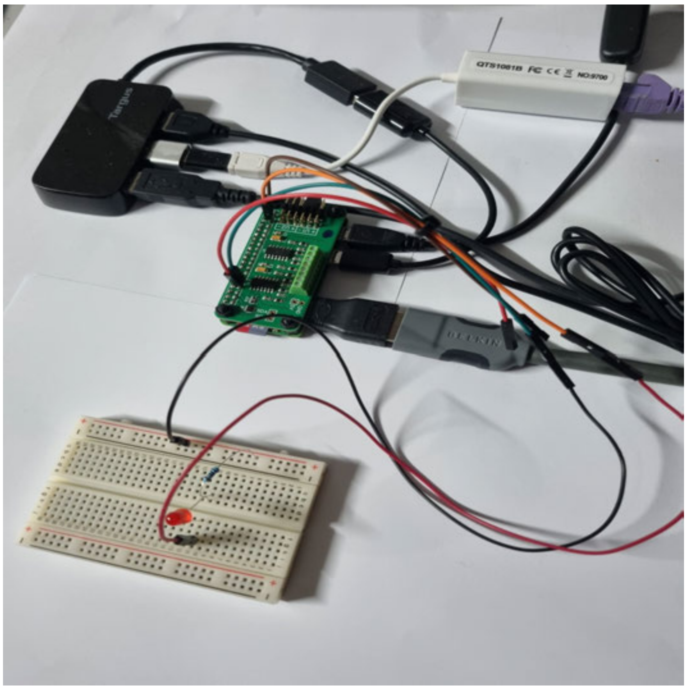


Figure 4.3: Image of the hardware connections to one of the Pi Zero units.

As detailed above, a 5-way USB Hub was connected via the Micro-USB adaptor to expand allowable USB ports for each Pi- Zero unit. The 5-way USB-A Hub subsequently had the below items connected:

- 1 x Ethernet RJ45 adaptor to Switch (USB-A to USB-C to USB-Micro in-line connections)
- 1 x USB Mouse connection
- 1 x USB Keyboard connection

For testing of I/O capability a small breadboard was used to connect LED's to the GPIO pins. This allowed visual tests to confirm discrete operations were working for each Pi Zero. Further connections to the 8-pin connection on the ADC module allowed for analog inputs to be connected as well. These elements are detailed further within the Results section of this document.

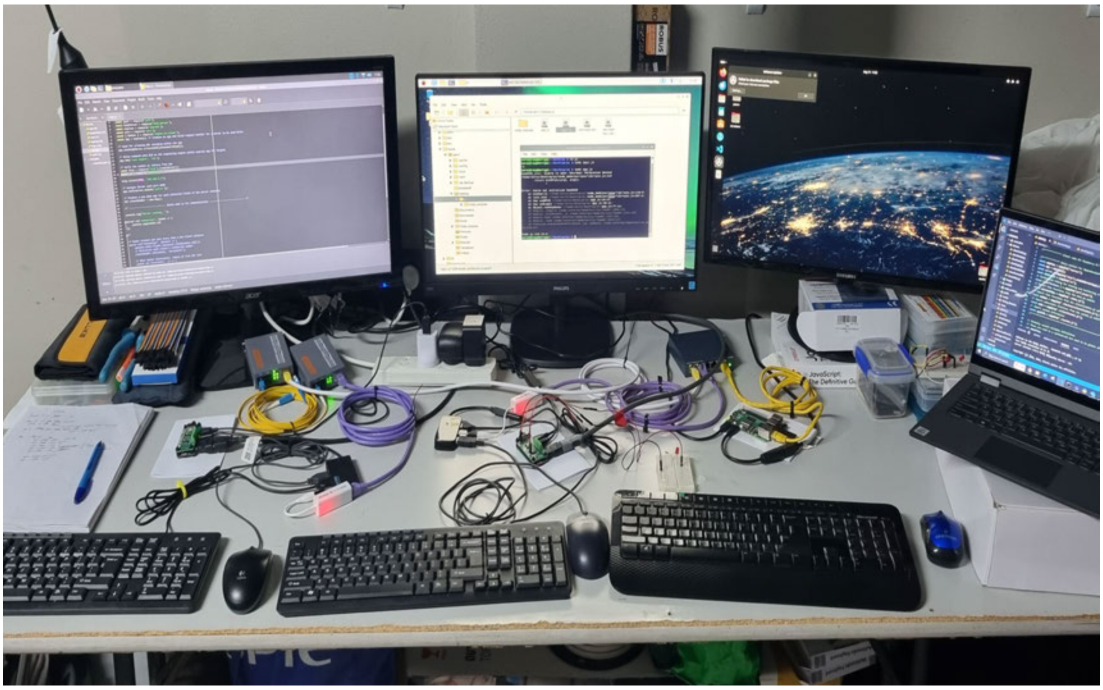


Figure 4.4: Image of the final system hardware setup.

4.2.3 Hardware - Material Take-Off (MTO)

A full list of all hardware and equipment used for construction of this project is listed in the below MTO table.

Description	**Model**	**Quantity**
Raspberry Pi Five	4GB Model B	1
Raspberry Pi Zero W	512MB Wireless	2
ADC Pi Module	ADC Pi	2
32GB Micro-SD Card	Sandisk	3
Raspberry Pi Five 5A Power Supply Unit	Pi Five PSU	1
5VDC USB-A Charger	Generic	2
HDMI Monitor (with power adaptor)	Generic	3
Keyboard	Generic	3
Mouse	Generic	3
USB-A 5-way Hub	Targius	2
Micro-USB to USB-C Adaptor	Generic	2
USB-C to USB-A Adaptor	Generic	2
Micro-USB to USB-A Adaptor	Generic	2
Ethernet to Micro-USB Adaptor	Generic	2
Ethernet to Fibre Media Convertors	NetLink (Pair)	1
5-Port Unmanaged Network Switch	TP-Link	1
Cat-6 Network cable	3m length	4
Single Mode LC Fibre Pair	3m Length	1
Standard HDMI cable	2m length	3
Micro HDMI to Standard Adaptor	Generic	1
Mini HDMI to Standard Adaptor	Generic	2
IRF520 Mosfet PWM Module	Pack of 5	1
4-Gang Powerboard	Generic	2

Table 4.1: Material Take-Off for project equipment

4.2.4 Test Equipment

A full list of equipment used for testing during this project is listed in the below table.

Description	**Model**	**Quantity**
Digital Multimeter	Fluke 289	1
Voltage / Current Calibrator	MANN Portacal 1000	1
Oscilloscope	PicoScope 2204A	1
Laptop	Lenovo	1
Test Leads & Plug Connectors	Generic Set	1

Table 4.2: Test equipment used within project

4.3 Software Design and Development

The software for each of the separate MCU's was developed as a NodeJS / Express runtime application which executed on top of the MCU operating system. Each board's main application could be invoked simply using a terminal command once, within the parent directory of the developed solution. Further information relating to the initial installation of operating systems, NodeJS runtime and application initialisation are provided in Appendix D.

The following section details software design following these initial installations through to full development of the final project code. The following subsections include:

- Pi Five Server - Program Architecture
- Pi Zero Clients - Program Architecture
- Network Communications and WebSockets Operations
- Sequential and Time-Based Control within Server / Client Applications
- Database Operations
- Model-View-Controller - HMI display
- System Information and Data Monitoring

4.3.1 Pi Five Server - Program Architecture

The Pi Five server acted as the central point for the overall distributed control system with the core functionality of establishing communications with clients, monitoring alarms, performing database operations and providing HMI display data for the user. To achieve this functionality, a number of bespoke functions were developed as part of the server application. The software flow of the server.js application follows the below sequence.

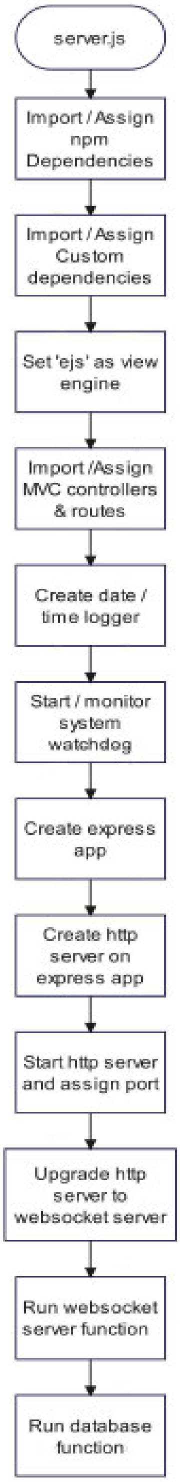

server.js
Import / Assign npm Dependencies
Import / Assign Custom dependencies
Set 'ejs' as view engine
Import /Assign MVC controllers & routes
Create date / time logger
Start / monitor system watchdog
Create express app
Create http server on express app
Start http server and assign port
Upgrade http server to websocket server
Run websocket server function
Run database function

Figure 4.5: Server software sequence flowchart.

The steps from the above flowchart are further detailed here:

1. **Import / Assign npm Dependencies**: This step imported and assigned all required npm library package dependencies into the application. This allowed the use of express, websockets, path and the body-parser packages.

2. **Import / Assign Custom Dependencies**: This step imported and assigned all custom made function dependencies into the application. These included the date-time, watchdog, serversocket and database functions created for the project.

3. **Set ejs as view engine**: This assigned EJS as the view engine for serving HTML webpage content. This allowed improved data and page functionality in the MVC operation.

4. **Import / Assign MVC controllers & routes**: This linked the controllers used for serving webpage content and defined the routes used to render the content. This is explained further in the HMI Section 4.3.10.

5. **Start / Monitor system watchdog**: The system watchdog started when the server is brought online. The watchdog refresh periodically reset the watchdog to ensure the system did not hang during computation. This is explained further in Section 4.3.7.

6. **Create express app**: The app returned in this command was a JavaScript function which can be passed to the NodeJS HTTP server as a callback. This allowed passing of the express request function handler which controls web-based HTTP request routing and rendering HTML responses. It also allowed assignment of middleware such as EJS (MDN 2024).

7. **Create HTTP server on express app**: This step generated a HTTP server and passed the express app function handler to the server for functionality.

8. **Start HTTP server and assign port**: This step simply created a HTTP server instance and listens on the assigned PORT 3000.

9. **Upgrade HTTP server to WebSocket server**: This step upgraded the HTTP server to a WebSocket server on the same PORT and address.

10. **Run websocket server function**: This ran the server side WebSockets function wsServer() imported from serverSocket.js. This was responsible for managing all communication between clients including communication watchdogs, device I/O and system data, and alarm management. It also assisted in routing incoming data into the database. This is detailed in Section 4.3.5.

11. **Run database function**: This step initiated the MySQL database using myDB() within the Pi Five. It ensured the database was online, tables were generated and all data handling was managed. This also ran a local asynchronous setInterval function which logged system data such as CPU loading and memory into the database. These points are detailed more in Section 4.3.9.

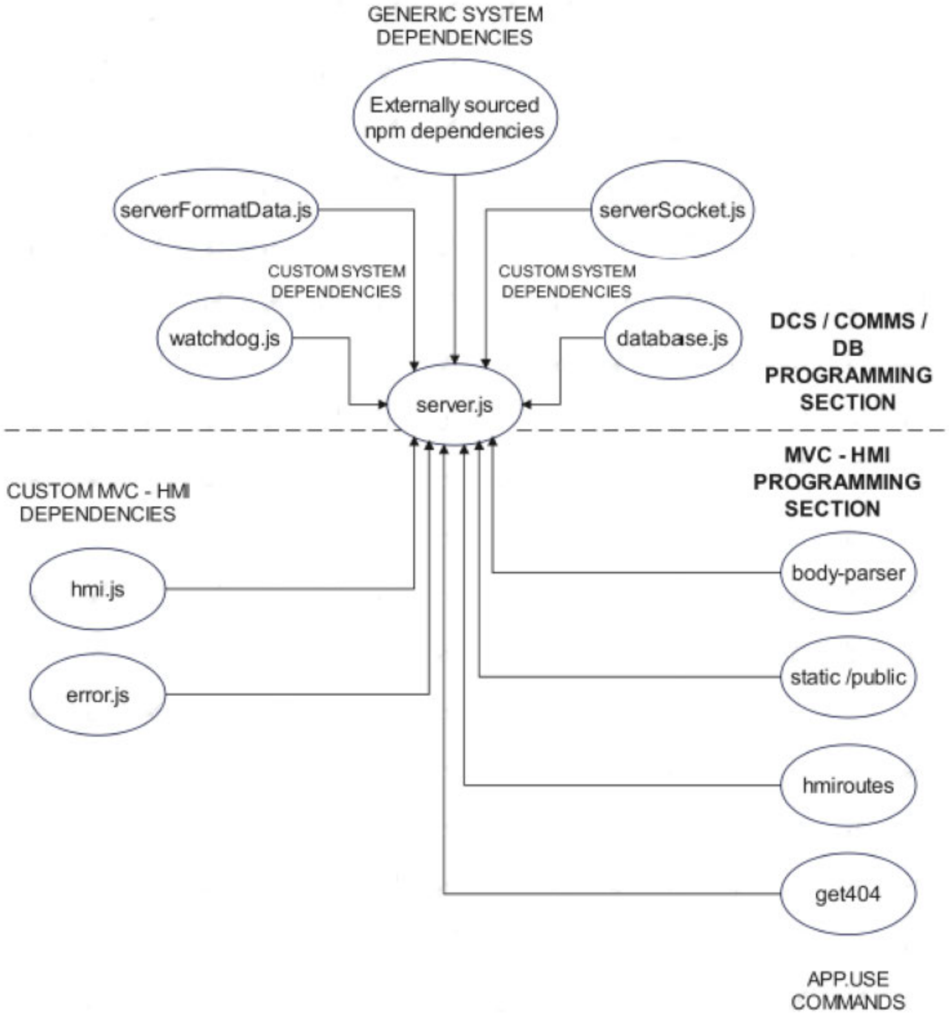

GENERIC SYSTEM DEPENDENCIES
Externally sourced npm dependencies
serverFormatData.js
serverSocket.js
CUSTOM SYSTEM DEPENDENCIES
CUSTOM SYSTEM DEPENDENCIES
watchdog.js
database.js
DCS / COMMS / DB PROGRAMMING SECTION
server.js
MVC - HMI PROGRAMMING SECTION
CUSTOM MVC - HMI DEPENDENCIES
hmi.js
body-parser
error.js
static /public
hmiroutes
get404
APP.USE COMMANDS

Figure 4.6: Overview of the system dependencies for the Server architecture.

4.3.2 Pi Zero Clients - Program Architecture

The architecture and program flow for the two separate Pi Zero W distributed control system clients were designed essentially identical except in two aspects. They were assigned a separate client number for communication and had different simulated I/O data created for testing purposes over the WebSockets connection, and into the server database. While hardware interfacing was tested (Section 5.3.6) the majority of the development for the client boards used simulated data to create larger data streams, more test points and assist with alarm activation tests.

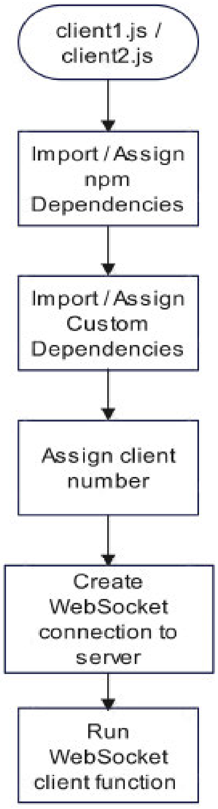

client1.js / client2.js
Import / Assign npm Dependencies
Import / Assign Custom Dependencies
Assign client number
Create WebSocket connection to server
Run WebSocket client function

Figure 4.7: Client software sequence flowchart.

While the control of field devices for processes using in-built algorithms is a major focus for individual embedded control systems, the function of these clients programs focused more specifically on distributed data control and alarm management. This required the ability to gather all updated device and system data, including device, alarm and system states, and communicate the data flows to the server and ultimately the database. In making the system robust, error checking and time-stamping functions were built into the communications to mimic industrial communication protocol qualities.

The steps from the above flowchart are further detailed here:

1. **Import / Assign npm Dependencies**: This step imported and assigned the required npm library package dependencies into the application. This allowed use of WebSockets package.

2. **Import / Assign Custom Dependencies**: This step imported and assigned all custom made function dependencies into the application. These included the client 1 & 2 I/O data functions and client WebSockets wsClient() function created for the project.

3. **Assign client number** : This simply assigned a number variable for passing into the wsClient() function for each client program. This was used to ensure watchdog and message functions were specific to each identified client.

4. **Create WebSocket connection to server**: This defined and created a connection point for passing into the wsClient() function.

5. **Run WebSocket client function**: This ran the wsClient() function, passing the connection, client number and a device I/O function updateIO() for use. The WebSocket wsClient() function is detailed in Section 4.3.5.

The client1.js and client2.js programs were simple in top level flow as the function calls were nested deeper within the wsClient() function. The updateIO() function was assigned and passed two levels down into the WebSockets function calls, with others also invoked in a similar style. These are explored in the following section.

4.3.3 Network Communications and WebSockets Operations

Initially the Socket.io, NodeJS library was selected for use within the project, however during development it was found that this library was not a true WebSockets implementation and while it offered certain benefits it also imposed some restrictions. These revolved around the ability to make low-level custom functions on top of the library definitions and added latency due to a higher-level implementation. Therefore, with a small impact on time resource the decision was made to move to the NodeJS 'ws', pure WebSockets library.

The first stage of network setup involved setting and confirmation of the Layer 3 - Network Layer, via a series of Ping tests on machine terminal sessions. The results of these tests are detailed in Section 5.2.2. Once these tests were confirmed the WebSockets (WS) software within each device's NodeJS application could be developed upon.

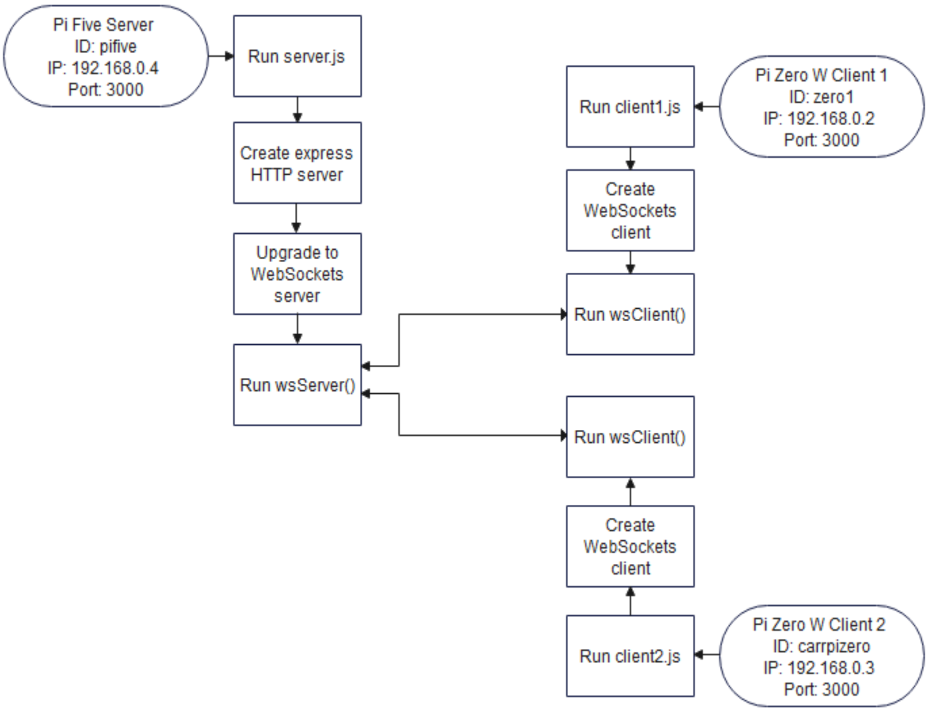


Figure 4.8: High-Level WebSockets network diagram

With the NodeJS 'ws' package installed, and imported into the npm initialised, base-level application, a basic structure for the WS could be built. This basic structure was taken direct from the npm library webpage at 'www.npmjs.com/package/ws' (Npmjs. 2024*b*), which lists various implementations for the package, including how to create & destroy an instance, send and receive messages, catch errors and action events.

Server WebSockets function - wsServer()

The server was the central data exchange for the distributed system and had some specific routines which ran to serve this purpose. Once the Pi Five's main NodeJS server.js application was running, the HTTP server was upgraded to a WebSockets server and the program called the wsServer() function. At this point the WebSockets server, which was assigned as a constant, was passed as a parameter to the wsServer() function. This allowed the WebSockets library to be called within the parent function and was a useful strategy for adding functionality while encapsulating the function itself.

The major functions are listed below and on the following page as a flow chart.

- Check for connection - Built-in WS function detects if client connected to the specified Port.
- Executes serverWatchdogInitiate(). This sends msg.type = 'wd' to the client which prompts the client to return it's unique ID and a trigger msg.type = 'wdinit' to start the server watchdog.
- Logs that a client has connected (for information and testing only).
- Opens an event.listener for messages. Built-in function which allows messages to be detected and logic to be developed for individual message and data types.
- If a message is received, this triggers the custom function msg(data) which takes the message data, confirms the type and assigns an action. These operations are detailed further below.
- If the WS connection is closed it logs whether the closure was requested or terminated without reason.
- If an error is detected using standard library functions it is simply logged at this stage.

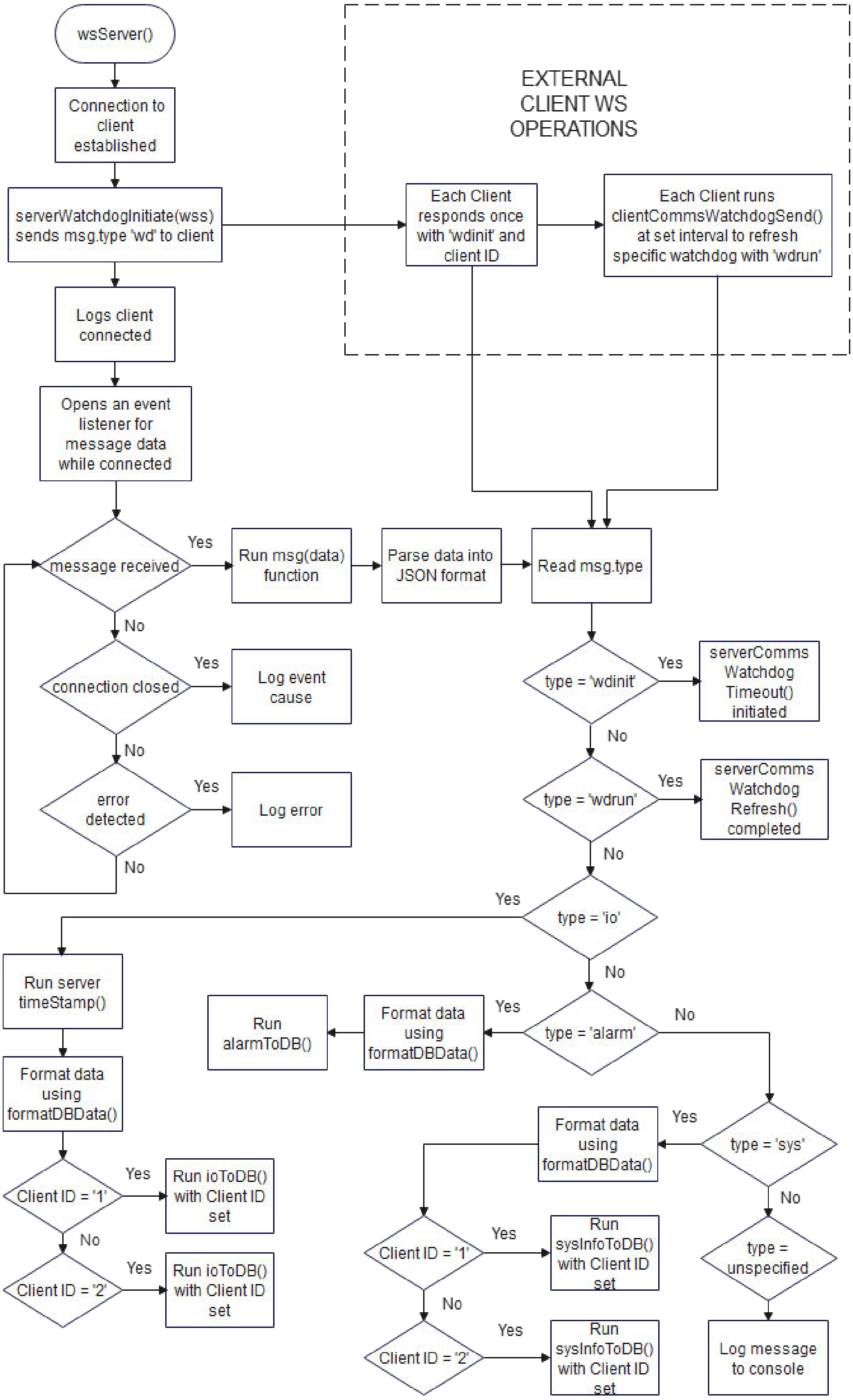

wsServer()
Connection to client established
serverWatchdogInitiate(wss) sends msg.type 'wd' to client
Logs client connected
Opens an event listener for message data while connected
EXTERNAL CLIENT WS OPERATIONS
Each Client responds once with 'wdinit' and client ID
Each Client runs clientCommsWatchdogSend() at set interval to refresh specific watchdog with 'wdrun'
message received
Yes
Run msg(data) function
Parse data into JSON format
Read msg.type
No
connection closed
Yes
Log event cause
No
error detected
Yes
Log error
No
type = 'wdinit'
Yes
serverComms Watchdog Timeout() initiated
No
type = 'wdrun'
Yes
serverComms Watchdog Refresh() completed
No
type = 'io'
Yes
Run server timeStamp()
Format data using formatDBData()
No
type = 'alarm'
Yes
Format data using formatDBData()
Run alarmToDB()
No
type = 'sys'
Yes
Format data using formatDBData()
No
type = unspecified
Log message to console
Client ID = '1'
Yes
Run ioToDB() with Client ID set
No
Client ID = '2'
Yes
Run ioToDB() with Client ID set
Client ID = '1'
Yes
Run sysInfoToDB() with Client ID set
No
Client ID = '2'
Yes
Run sysInfoToDB() with Client ID set

Figure 4.9: Pi Five WebSockets Server software sequence flowchart.

Server message function - msg()

As a message was detected using the standard WebSockets ‘ws’ library, the custom msg() function was called and used to process incoming data, determine the type and assign actions. This was the main internal function of wsServer(), responsible for calling further custom functions to complete actions. The main operations can be seen in Figure 4.14

- Parses incoming data into a JavaScript Object Notation (JSON), a lightweight text-based format for processing data.
- Checks the message type using msg.type function and runs code specific to each case.
- For type = ’wdinit’, initiates a new instance of serverCommsWatchdogTimeout(). This starts a timeout with the client ID attached with the ’wdinit’ command to ensure each client has a unique timer check for communications state.
- For type = ’wdrun’, calls the serverCommsWatchdogRefresh() function which resets the unique client watchdog timer ensuring the communications is polled, active and healthy. This is sent at a defined and adjustable period set within the client function.
- For type = ’io’, runs a custom timeStamp() function which assign the current data and time to a server time parameter. It then takes the received data and formats into a database ready configuration using another custom function formatDBData(). After this it checks the client ID sent with the data and passes this into another custom function ioToDB(), which writes the data into the appropriate MySQL database table. This data is the discrete and analog input / output data from each client.
- For type = ’alarm’, the data is again formatted using formatDBdata() then written into the database alarm table using custom function alarmToDB(). This data represents alarms generated on either client.
- For type = ’sys’, again, formatDBdata() is run. Following this, the client ID is checked and custom function sysInfoToDB() is called which writes the specific client system information into its assigned database table.
- For any unspecified messages the data is simply logged to console for user information at this stage.

Client WebSockets function - wsClient()

Once either Pi Zero W client connected via its WebSockets instance, it called the custom wsClient() function. Again, this function had a series of further custom internal functions which executed to provide functionality. The wsClient() function was passed three parameters which were the WebSockets client instance, the client number assigned in the parent client1/2.js main NodeJS function, and the updateIO() function.

The major functions are listed below and on the following page as a flow chart.

- Runs an Event Listener for an open socket event. If detected it simply logs the server connection for information.
- Runs an Event Listener for message data. This calls another custom client side msg() function. This function simply checks for type ‘wd’, and sends back a msg type ‘wdinit’, with the client ID that is passed as a parameter to wsClient(). If the message is unspecified with no type, it simply logs to console for information at this stage.
- If the WS connection is closed it logs whether the closure was requested or terminated without reason.
- If an error is detected using standard library functions it is simply logged at this stage.
- Next, clientCommsWatchdogSend() is called. This starts a re-occuring, setInterval operation which calls function clientCommsWatchdogUpdate() at the specified interval time. The basic task of this is to reset the decrementing watchdog setTimeout operation.
- Next, sendData() is called. This starts a re-occuring, setInterval operation which calls clientSendData(). This further calls function updateIO() which returns all current analog and discrete input / output device values and alarm status. Once returned within clientSendData(), this data is timestamped using timeStamp(). Then the data is converted into a discrete and analog device data map and converted to JSON format. Alarms are then one-shot verified to avoid flooding at the server end and the data is sent to the server for processing and database storage.

- Lastly, sendSystemData() is called. This starts a re-occuring, setInterval operation which calls sysInfo(). This further calls createsysMap() which pulls local operating system status information and creates a data map of the data. It then converts to JSON format and sends to the server.

The following page details the program flow for the developed wsClient() function.

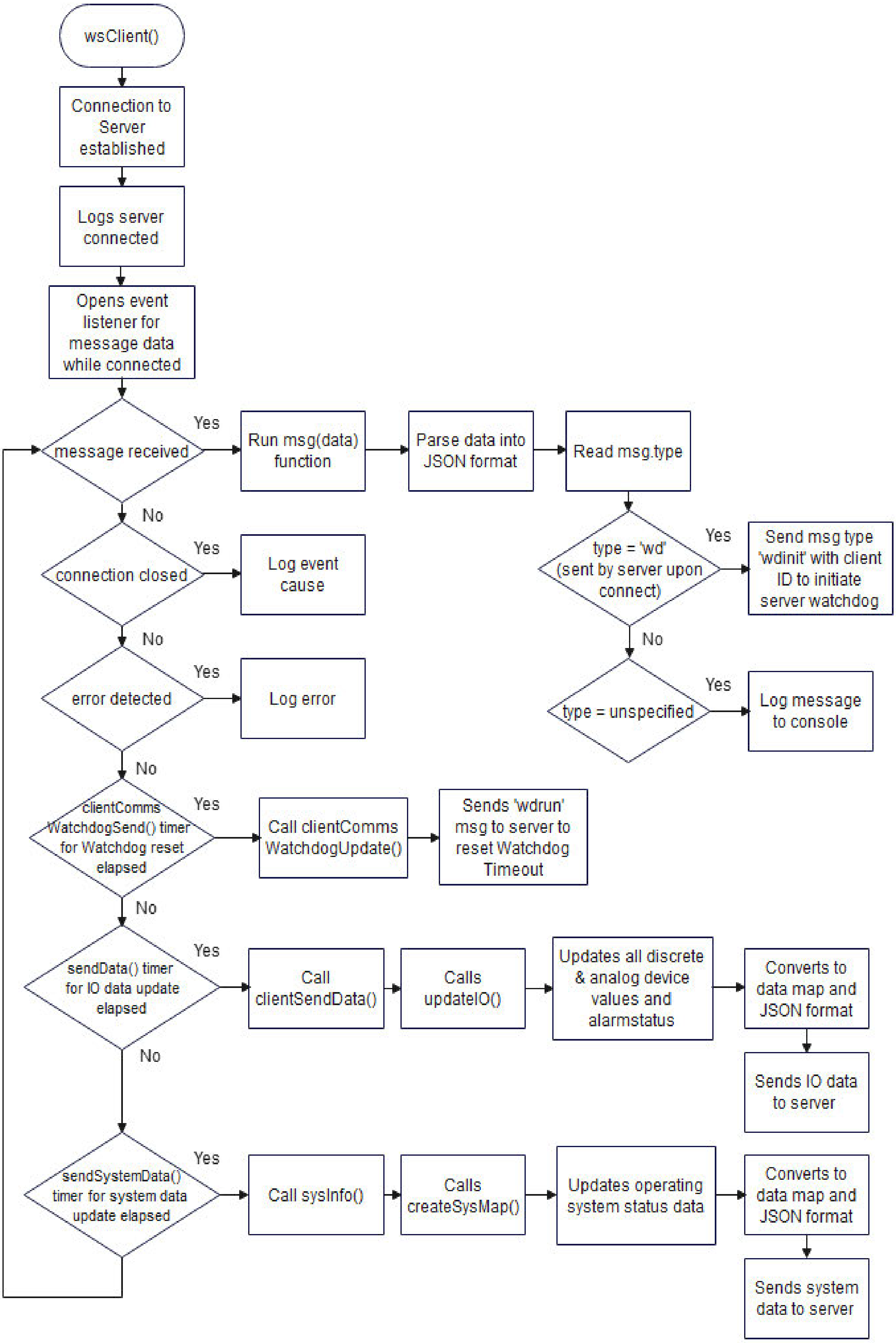

wsClient()
Connection to Server established
Logs server connected
Opens event listener for message data while connected
message received
Yes
No
Run msg(data) function
Parse data into JSON format
Read msg.type
type = 'wd' (sent by server upon connect)
Yes
Send msg type 'wdinit' with client ID to initiate server watchdog
No
type = unspecified
Yes
Log message to console
connection closed
Yes
Log event cause
No
error detected
Yes
Log error
No
clientComms WatchdogSend() timer for Watchdog reset elapsed
Yes
Call clientComms WatchdogUpdate()
Sends 'wdrun' msg to server to reset Watchdog Timeout
No
sendData() timer for IO data update elapsed
Yes
Call clientSendData()
Calls updateIO()
Updates all discrete & analog device values and alarmstatus
Converts to data map and JSON format
Sends IO data to server
No
sendSystemData() timer for system data update elapsed
Yes
Call sysInfo()
Calls createSysMap()
Updates operating system status data
Converts to data map and JSON format
Sends system data to server

Figure 4.10: Pi Zero W WebSockets Client software sequence flowchart.

Communications Watchdog functions and operation

The server and client communications were monitored by a watchdog process which accepted individual client numbers and performed periodic checks to ensure connectivity was maintained. This was a core function to ensure reliable network communications were maintained with the aim of replicating industrial robustness. The custom functions created for this purpose were held within the watchdog.js file and exported to the necessary WebSockets server and client functions. These are detailed below and numbered according to the flow diagram on the following page.

1. **serverWatchdogInitiate(socket)**

   This function is invoked upon detection of a connection to a client and simply sends a socket message of type: ‘wd’ with ID: ‘server’ to the newly connected client. The WebSockets instance is passed in to allow the send function to be performed internally.

2. **serverCommsWatchdogTimeout(socket, clientID)**

   This is invoked when message type: ‘wdinit’ is received by the server. It takes the client ID and assigns it as the commsTimer[ID] for a setTimeout() rundown function which terminates the socket instance also passed into the function, if not reset by the below function within the given time period.

3. **serverCommsWatchdogRefresh(clientID)**

   This is invoked when message type: ‘wdrun’ is received by the server. It takes the clientID which is also within the message frame and uses this to set a clearTimeout(commsTimer[ID]) operation to reset the decrementing watchdog timer specific to the client.

4. **clientCommsWatchdogSend(WDflag, socket, clientNumber)**

   This is called by the client at the initial WebSockets connection instance. This starts a re-occuring setInterval operation which calls function clientCommsWatchdogUpdate() at the specified interval time. The basic task of this is to send msg type ‘wdrun’ to the server to reset the decrementing watchdog setTimeout operation as stated in the above definitions. The WDflag ensures it is called after the ‘wdinit’ command has been sent to the server ensuring the watchdog is already initiated.

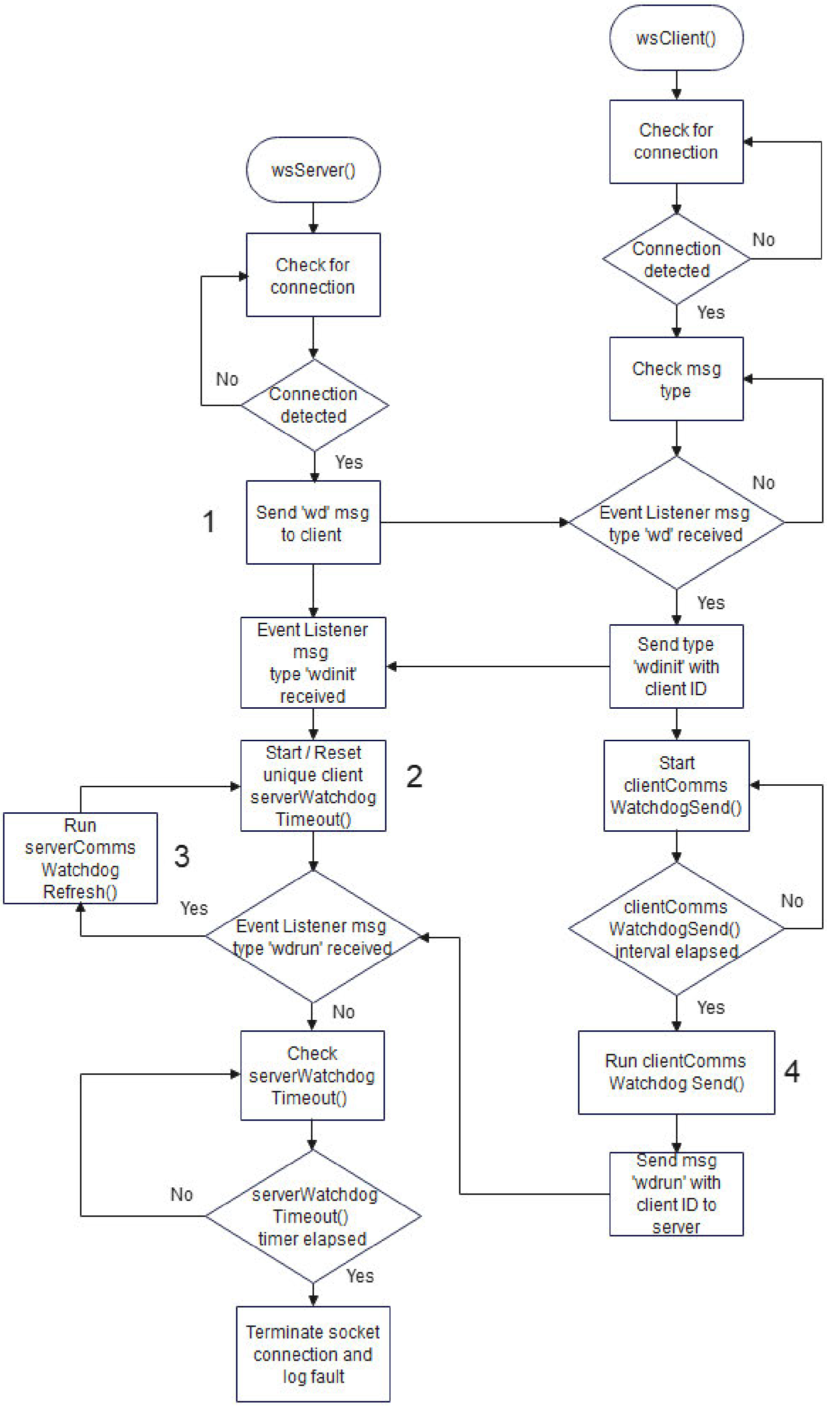

wsClient()
Check for connection
Connection detected
No
Yes
Check msg type
Event Listener msg type 'wd' received
No
Yes
Send type 'wdinit' with client ID
Start clientComms WatchdogSend()
clientComms WatchdogSend() interval elapsed
No
Yes
Run clientComms Watchdog Send()
4
Send msg 'wdrun' with client ID to server
wsServer()
Check for connection
No
Connection detected
Yes
1
Send 'wd' msg to client
Event Listener msg type 'wdinit' received
Start / Reset unique client serverWatchdog Timeout()
2
Run serverComms Watchdog Refresh()
3
Yes
Event Listener msg type 'wdrun' received
No
Check serverWatchdog Timeout()
No
serverWatchdog Timeout() timer elapsed
Yes
Terminate socket connection and log fault

Figure 4.11: Communications Network Watchdog software sequence flowchart.

4.3.4 Sequential and Time-Based Control within Server / Client Applications

To provide a workable distributed system the combination of sequential and time-dependent operations was needed. To achieve this a host of asynchronous functions and in-built JavaScript and NodeJS operations were utilised. These allowed functionality within key aspects such as watchdog timers, date and time currency, device and system data periodic aggregation and importantly in the execution of alarm delays within the developed analog device class aiDevice.

The asynchronous functions used for these cases were:

- **setInterval()** - This runs a repeating specified callback function at the set time interval in mS. This is useful for communication polling, periodic device status scans etc.
- **setTimeout()** - This specifies a callback function to execute in the future in mS. This can be used for watchdog timeouts, events and alarm activations.
- **Promise()** - This is used to ensure sequence of asynchronous actions. The below '.this' callback is the response when the promise has been resolved. This is an important function for ensuring order of operations and is used within the alarm activation of aiDevice's method checkAlarm() method.
- **this. (promise callback)** - As stated, this callback is the trigger to inform dependent processes that a certain pending operation has been completed, critical for maintaining system order and function.

(NodeJS. 2024)

Many of the re-occurring, time-dependent, sequenced operations within the applications utilised the above asynchronous functions. These were straight forward implementations for operations such as cyclical polling, watchdog resets and the like, shown in the list below.

- sysWatchdog()
- sysWatchdogRefresh()
- serverWatchdogInitiate(socket)
- serverCommsWatchdogTimeout()
- serverCommsWatchdogRefresh()
- timeStamp()
- clientCommsWatchdogSend()
- sendData()
- sendSystemData()

Reading device values, discrete and analog state functions

Further more technical implementations of asynchronous operations exist within class methods of the aiDevice{} class including delayTimer() and setHHAlarmFlag() which were internal methods of the checkAlarms() method. These sections of program can be reviewed within the Appendix D Section D.4 and are detailed with the parent function updateIO()below.

updateIO()

This was the main interfacing and polling function used in the clients to allow cyclical polling of real and later, simulated devices for testing. The major steps performed by this function for each client are given below:

- Imports both the digital ioDevice() function and analog aiDevice class for use.
- Creates object instances of digital devices assigning GPIO scanned (or simulated) values as their input.

- Creates analog object instances by scanning ADC Pi analog voltage (or simulated) values on channels, then inputs the value with other parameters into the class constructor.
- Builds a combined digital / analog device data map.
- Calls the checkAlarm method on each device (analog only in project scope).
- Any devices with active alarms go into a separate alarm data map for sending to server/db.
- Interlocks alarm sending with one-shot flag to stop flooding.
- Converts device and alarm data to JSON for WS transmit to server.

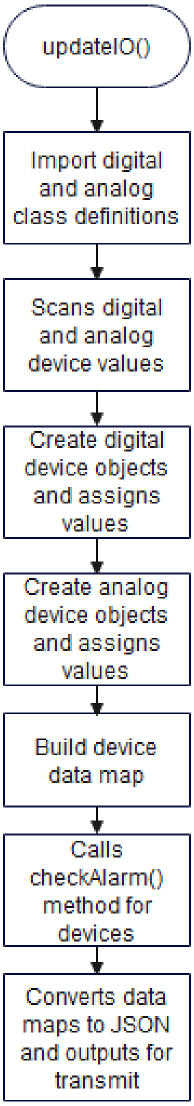

updateIO()
Import digital and analog class definitions
Scans digital and analog device values
Create digital device objects and assigns values
Create analog device objects and assigns values
Build device data map
Calls checkAlarm() method for devices
Converts data maps to JSON and outputs for transmit

Figure 4.12: updateIO() device scan software sequence flowchart.

aiDevice{}

This class was developed for use with all analog inputs used in conjunction with the client ADC Pi units. The class was called within the above detailed updateIO() function to create individual analog objects and assign the in-built methods for use, allowing such functionality as monitoring the specific device process variable and alarm status.

This class was critical in developing and testing the ability to create timed alarm delay activation methods. While only a High High test alarm was implemented owing to project time resource management, the test alarm operated successfully ensuring any subsequent future developed alarms can operate using the same code framework. It achieved this using promises and their associated callbacks to set alarm flags high after the predetermined delay period. This showcased the ability of NodeJS to create and initiate separate thread, timed execution callbacks within the overall application program loop running. The class is detailed below:

- Initial class constructor creates the object model with all parameters passed into the object upon instantiation.
- Getter method currentPV() allows scaledPV to be returned to program when called.
- updateRaw(raw) method allows the scaled PV to be updated on cyclical basis.
- resetAlarm(reset) method allows the alarm flag to be reset after activation.
- delayTimer(t, timeoutID) method creates a promise with an internal timeout responsible for activating the alarm delay upon expiration.
- setHHAlarmFlag() method encapsulated delayTimer() method and returns TRUE.
- checkAlarms() method uses the above delayTimer() and setHHAlarmFlag() with further instant alarm and healthy status check code sections to manage device alarms returning the scaled process variable, alarm flag, alarm data and tag ID.

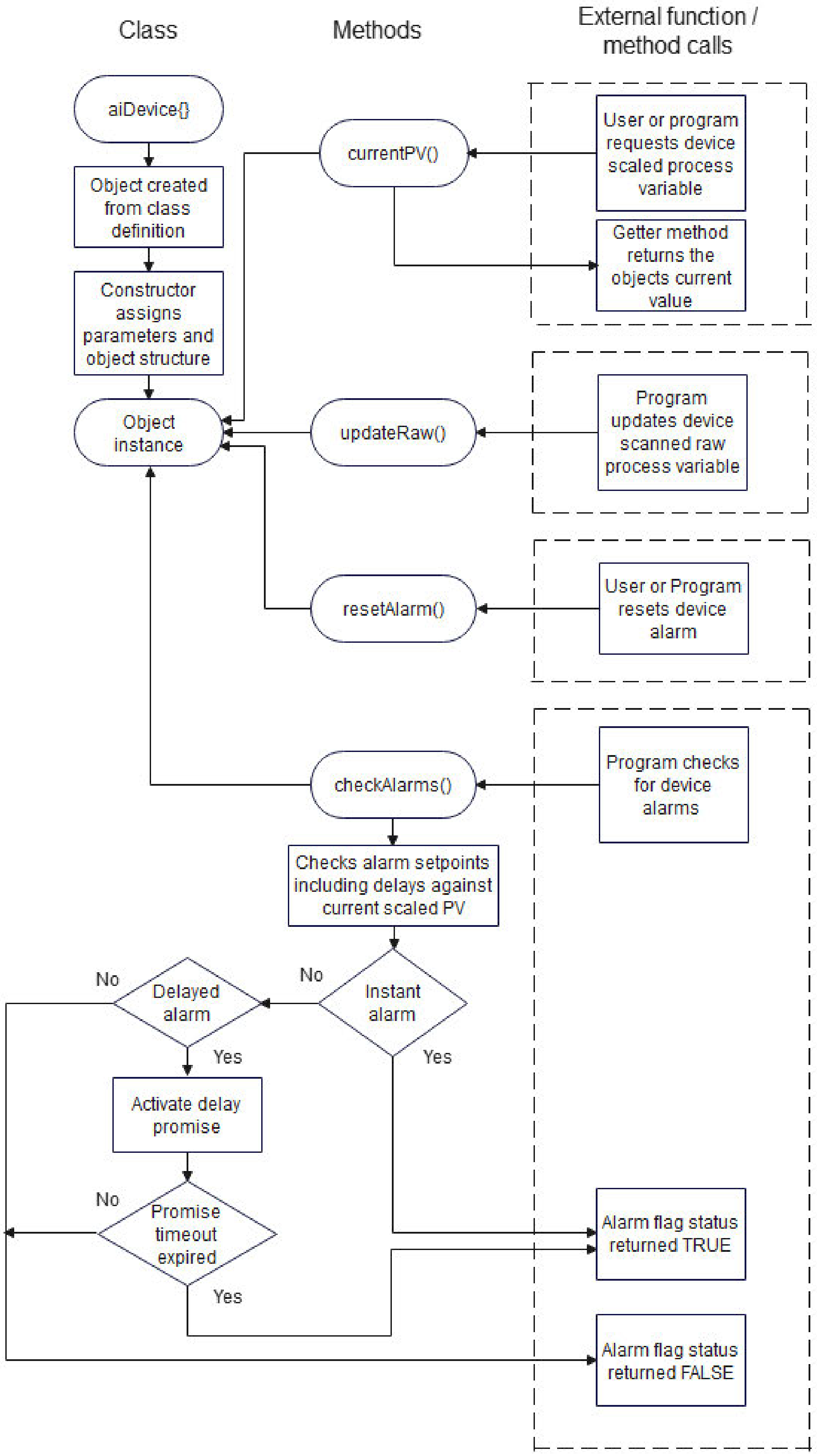

Class
Methods
External function / method calls
aiDevice{}
Object created from class definition
Constructor assigns parameters and object structure
Object instance
currentPV()
User or program requests device scaled process variable
Getter method returns the objects current value
updateRaw()
Program updates device scanned raw process variable
resetAlarm()
User or Program resets device alarm
checkAlarms()
Program checks for device alarms
Checks alarm setpoints including delays against current scaled PV
Instant alarm
No
Yes
Delayed alarm
No
Yes
Activate delay promise
Promise timeout expired
No
Yes
Alarm flag status returned TRUE
Alarm flag status returned FALSE

Figure 4.13: aiDevice() class software flowchart.

clientSendData()

This function was developed in conjunction with the other device interface code and was responsible for handling the updateIO() function which was passed as a parameter. This function was called within the clientSocket() function and takes the client ID, socket instance and updateIO() function as parameters. It has an adjustable poll interval which was used to test the capability of the WebSockets connection with 300-500ms update rates, documented in the results Section 5.3. The major functional steps for this function are:

- Called within clientSocket() WebSockets function
- Calls the updateIO() function and assigns the returned IO data to a variable.
- Creates a timestamp and assigns to the IO and alarm data
- Checks for a one-shot, toggled alarm flag which is returned, if TRUE, the function transmits the logged alarm status. (This resets automatically within the updateIO() function).
- Allows adjustable time intervals to be set for the polling of client discrete and analog devices.

4.3.5 Database Operations

A key aspect of distributed control systems is the ability to store and manage information from various sources. To enable this functionality within the developed system, a compatible database system was setup and integrated with the application. This included the below steps:

- Download and Install the database software.
- Development of database interface software.

The first step is detailed in Appendix D, Sub-Section D.3.3, with further outcomes in the Results Sub-Section 5.3.2.

Development of database interface software

To enable database interaction via the application two custom files with multiple functions were created which allowed create, read, update and delete (CRUD) operations for associated data. The custom database.js file holds the functions myDB(), ioToDB(), alarmToDB() and sysInfoToDB() which are detailed below. The separate serverFormatData.js holds formatData(), a function capable of reformatting data into a key/value array pair which can be then be loaded into the relevant database tables for tag and value entries. Figure 4.18 on the following page displays the high-level database operations which allow device status, alarm status and system data to be logged in the database.

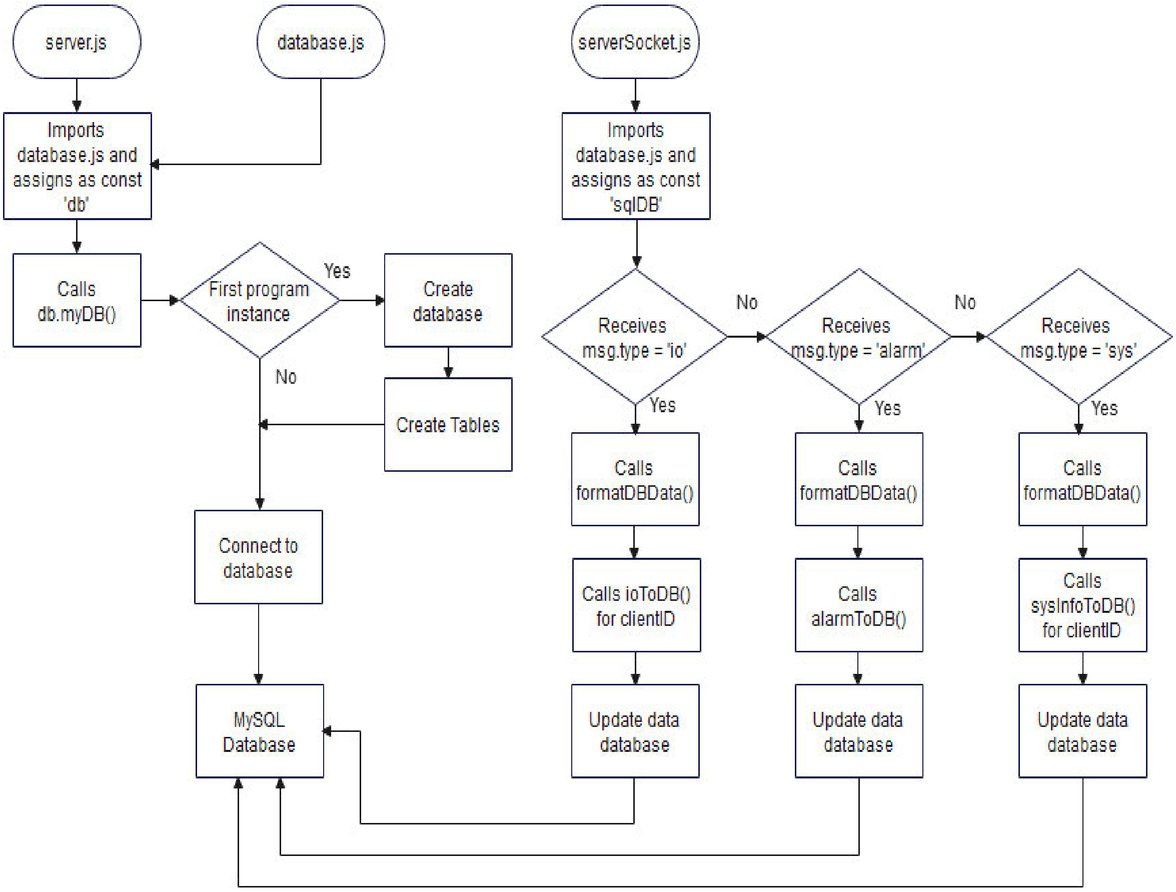


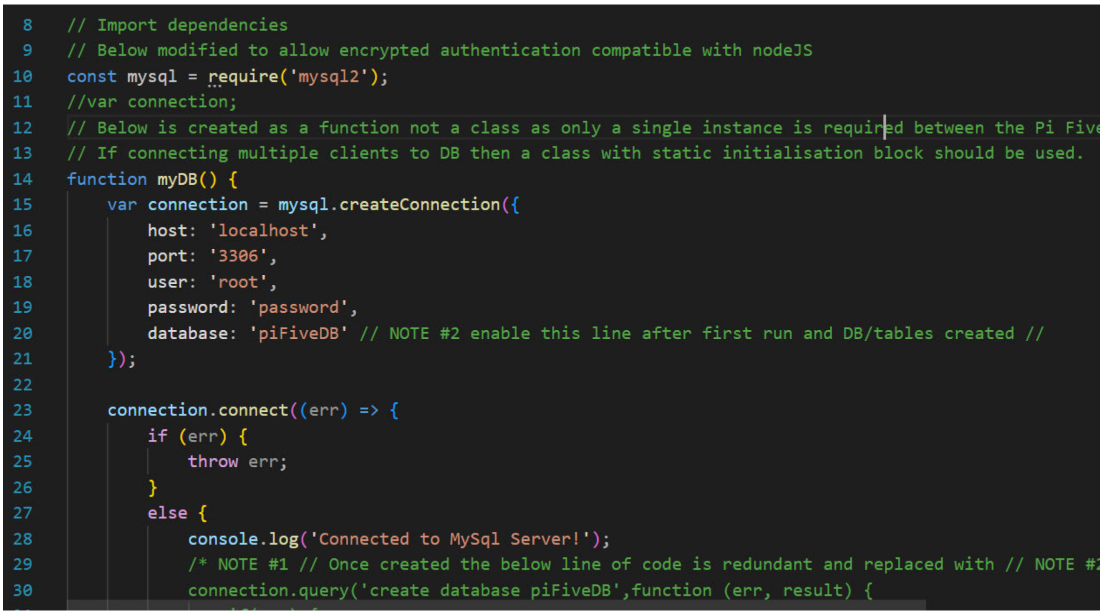
Figure 4.14: MySQL Database and NodeJS server application integration flowchart.

Database functions and operation

The custom functions created for interfacing between the MySQL database and NodeJS server application are listed below.

1. **myDB()**

   This function was responsible for initially creating the database instance and all tables. For the purpose of this project the task was manually performed once, with the sections then commented out. The remaining function simply connected to the database listed using the necessary password and port: 3306 as shown in Figure 4.19.



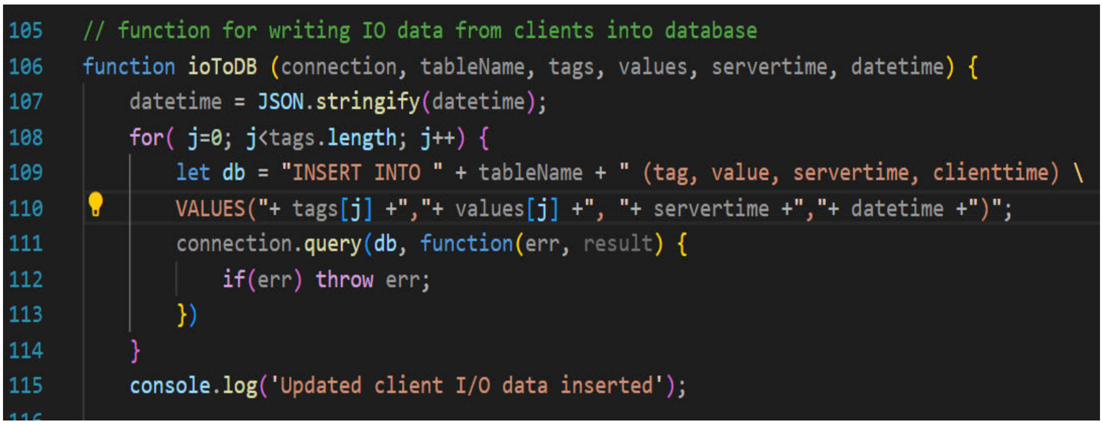
```
// Import dependencies
// Below modified to allow encrypted authentication compatible with nodeJS
const mysql = require('mysql2');
//var connection;
// Below is created as a function not a class as only a single instance is required between the Pi Five
// If connecting multiple clients to DB then a class with static initialisation block should be used.
function myDB() {
    var connection = mysql.createConnection({
        host: 'localhost',
        port: '3306',
        user: 'root',
        password: 'password',
        database: 'piFiveDB' // NOTE #2 enable this line after first run and DB/tables created //
    });

    connection.connect((err) => {
        if (err) {
            throw err;
        }
        else {
            console.log('Connected to MySql Server!');
            /* NOTE #1 // Once created the below line of code is redundant and replaced with // NOTE #2
            connection.query('create database piFiveDB',function (err, result) {
```

Figure 4.15: myDB() function screenshot.

2. **iotoDB(connection, tableName, tags, values, servertime, datetime)**

   This function accepted 6 parameters to enable it to process and write to the database. The connection was where the MySQL instance was passed into the function. Tablename specified client 1 or 2, the tags and values were pre-processed by the formatDBData and were ready to write as received. The servertime was taken prior to calling the function and datetime was the client side timestamp which allowed for latency determination.

The function checked the length of the tags array and performed a ‘for’ loop which iterated until all data in the WebSockets message was updated into the database table. The structure and operation for alarmtoDB() and sysInfoToDB() functions were highly similar to the ioToDB() function which is shown in Figure 4.20 for reference.



```
// function for writing IO data from clients into database
function ioToDB (connection, tableName, tags, values, servertime, datetime) {
    datetime = JSON.stringify(datetime);
    for( j=0; j<tags.length; j++) {
        let db = "INSERT INTO " + tableName + " (tag, value, servertime, clienttime) \
        VALUES("+ tags[j] +","+ values[j] +", "+ servertime +","+ datetime +")";
        connection.query(db, function(err, result) {
            if(err) throw err;
        })
    }
    console.log('Updated client I/O data inserted');
```

Figure 4.16: ioToDB() function screenshot.

3. **alarmToDB(connection, tableName, tags, values, datetime, clientID)**

   This function mirrored the operations for the above ioToDB() except with differing parameters and some JSON formatting operations. This function logged the tag, alarm type as a string, date / time and client in which the alarm originated from.

4. **sysInfoToDB(tableName, tags, values, datetime)**

   This function was also similar to the above two, taking the system data from either clients or server, time-stamping the data and iterating a loop which transferred all the data into the appropriately listed database table.

5. **ioDataFromDB()**

   This function was used as an example to prove the capability of the MVC to access the database and pull the most up-to-date device process variable for use on the HMI display. The function can easily be modified for future use to pull data from any device automatically to allow populating of the HMI displayed devices.

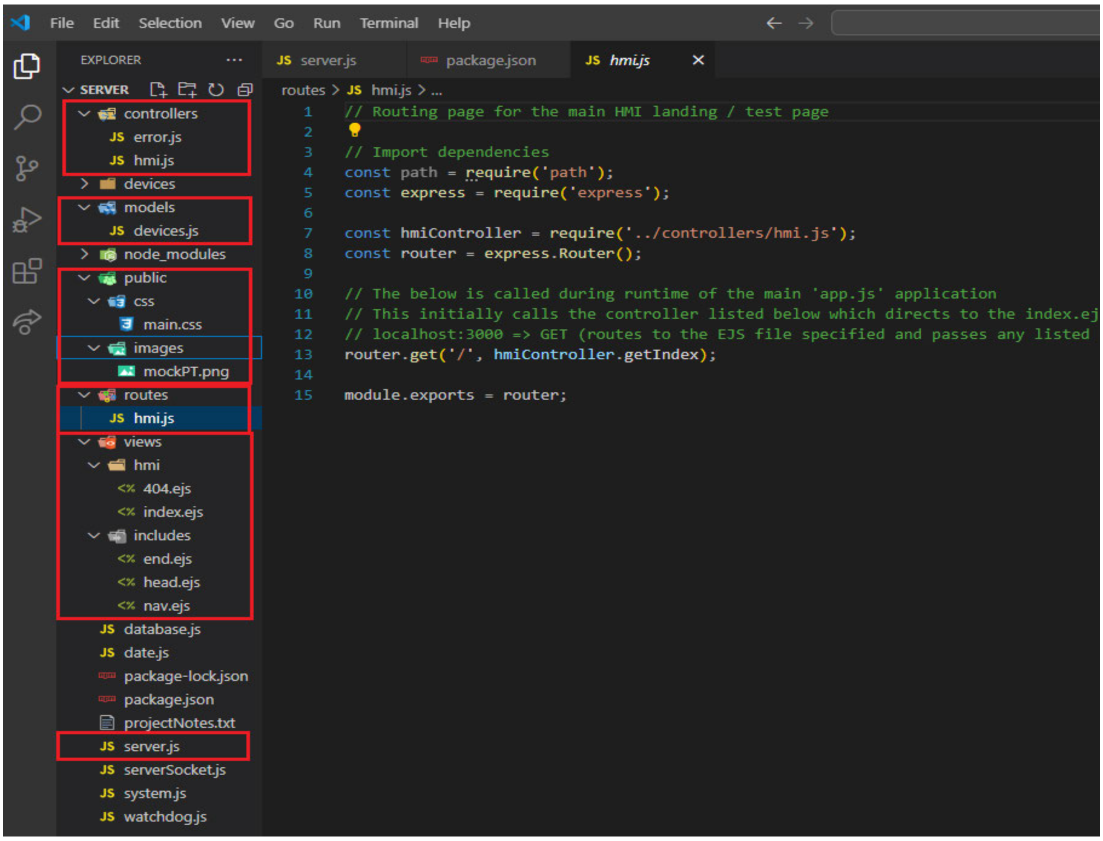
6. **formatDBData()**

   This function was held within the devices sub-folder in ‘serverFormatData.js’ and was responsible for formatting data ready for MySQL operations. It achieved this by parsing the received data, splitting into key/value pairs then creating to separate object arrays which were separately returned for use.

4.3.6 Model-View-Controller - HMI display

The Pi Five embedded board was the central server for the distributed system, therefore the server.js NodeJS program was responsible for processing the data and rendering the Human-Machine-Interface web page displays. To allow this, the Model-View-Controller structure (Hernandez 2021) was adopted, and used to explore how data from the application could be rendered within Hyper-Text-Markup (HTML) web pages. In addition, the EJS middleware package was utilised to assist with data references and creating templates for efficiency.

The organisation of an MVC approach allowed the application data to be efficiently moved between sections of the program and onto the HMI display. This process included a number of sections in which the data was called by functions and passed, until finally rendered as visual information. The MVC structure was first created in a file system held in the Visual Studio Code parent folder. This included sub-folders for models, views, controllers, routes and the public section.



```
// Routing page for the main HMI landing / test page

// Import dependencies
const path = require('path');
const express = require('express');

const hmiController = require('../controllers/hmi.js');
const router = express.Router();

// The below is called during runtime of the main 'app.js' application
// This initially calls the controller listed below which directs to the index.ej
// localhost:3000 => GET (routes to the EJS file specified and passes any listed
router.get('/', hmiController.getIndex);

module.exports = router;
```

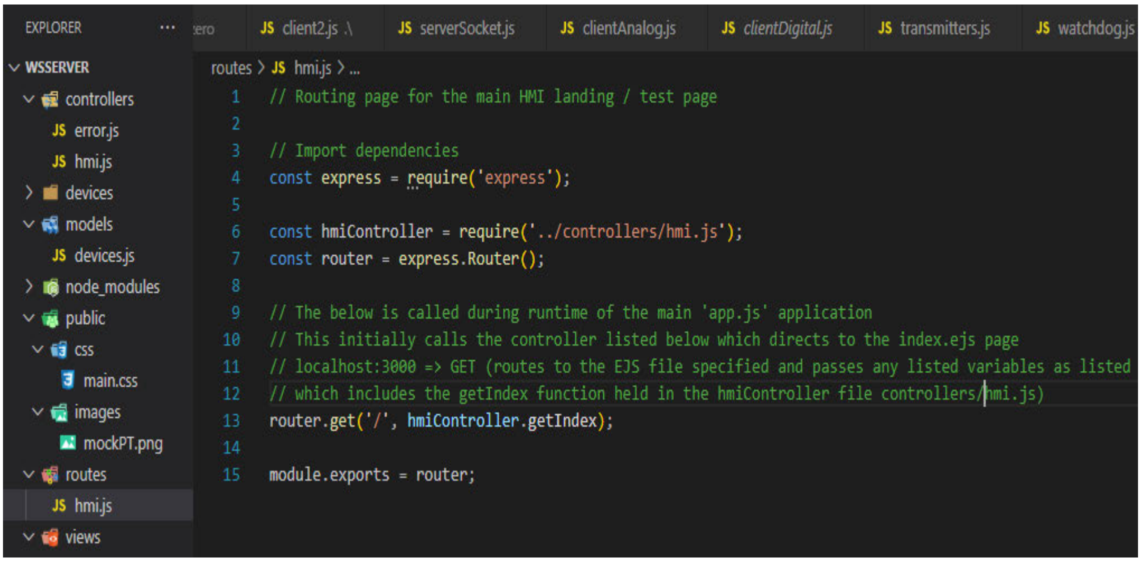
Figure 4.17: VS Code folder and file MVC structure.

The project's final MVC structure can be seen in the above Figure 4.21, with the following sections detailing how the contents of each folder allowed this system to function. While the project had a single page developed, an important characteristic of the MVC approach was that it could be easily extended for further pages and functionality in the future. Future additional sections can be added to reflect the flowchart displayed in the below Figure 4.22.

Routes

Initially the main 'server.js' NodeJS application file is executed which starts the MVC process by importing the needed dependencies and setting EJS as the middleware program. Following this, the application creates the HTTP express app and upgrades to a WebSockets server, however this function is reserved for communications, with the HMI-MVC process working from the HTTP level.

Once the server is created the express application uses the 'app.use' syntax to define the routes and controllers used within the MVC structure. For pages which are created for viewing with their associated routes listed, the MVC system will work to render the information onto the HMI. If an error is encountered, or a page is requested by the user which is not listed the error controller is called directly from the application bypassing the routing and serving the '404' standard error handling page.



```
// Routing page for the main HMI landing / test page

// Import dependencies
const express = require('express');

const hmiController = require('../controllers/hmi.js');
const router = express.Router();

// The below is called during runtime of the main 'app.js' application
// This initially calls the controller listed below which directs to the index.ejs page
// localhost:3000 => GET (routes to the EJS file specified and passes any listed variables as listed
// which includes the getIndex function held in the hmiController file controllers/hmi.js)
router.get('/', hmiController.getIndex);

module.exports = router;
```

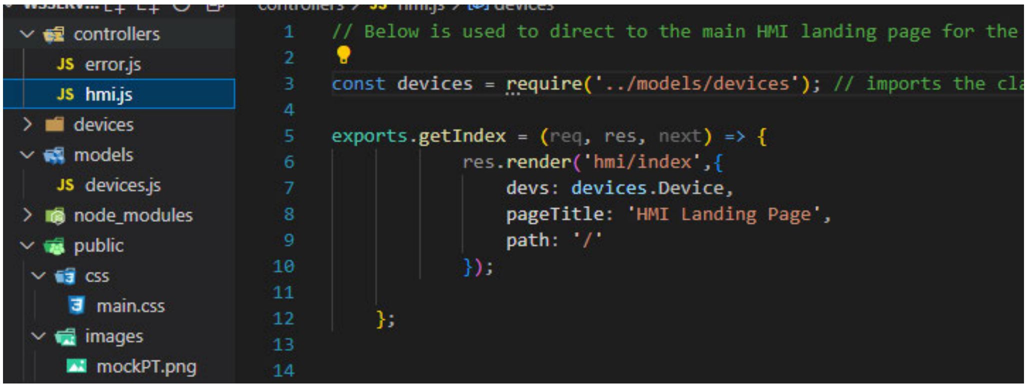
Figure 4.18: The HMI homepage router responsible for calling the index page and assigned controller

The ‘routes/hmi.js’ file matches the page trying to be accessed with its assigned controller, which, in this case assigns the ‘hmicontroller’ to the base index page marked as ‘/’ with its ‘.getIndex()’ function.

Controllers

There were two controllers created for the project, with both held within the controllers folder. These were the HMI homepage controller ‘hmi.js’ and error page controller ‘error.js’.

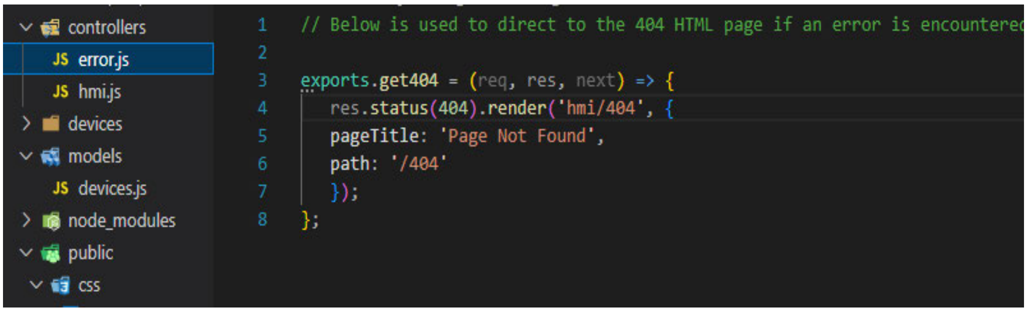
The HMI controller uses a JavaScript ‘GET’ request which is a core function that allows an object to be bound to a request. Using this method, the ‘getIndex(req, res, next)’ function calls the ‘views/hmi/index.ejs ’ file when the user types the ‘http://localhost:3000/’ address into the browser. It also calls the listed model ‘Device’ which is already imported within the ‘devices’ constant using the require method previously explained.



```
// Below is used to direct to the main HMI landing page for the

const devices = require('../models/devices'); // imports the cla

exports.getIndex = (req, res, next) => {
            res.render('hmi/index',{
                devs: devices.Device,
                pageTitle: 'HMI Landing Page',
                path: '/'
            });


    };
```

Figure 4.19: The HMI homepage controller

The error controller follows a similar process to render the ‘views/hmi/404.ejs’ page when an erroneous request is made. This is a simple handler, as in reality the user should be restricted to controlled page requests and was to aid development predominantly.

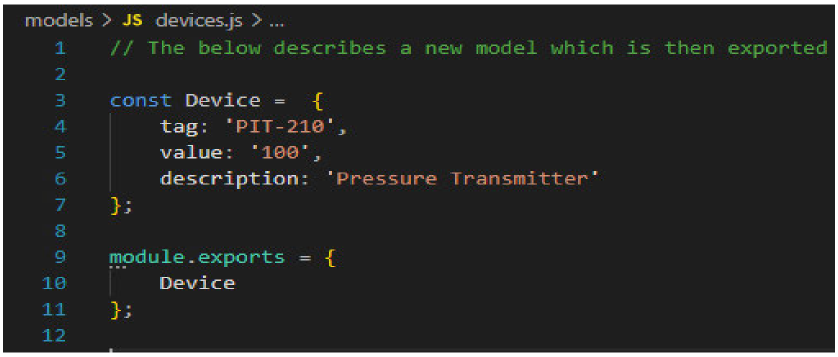


```
// Below is used to direct to the 404 HTML page if an error is encountere

exports.get404 = (req, res, next) => {
  res.status(404).render('hmi/404', {
  pageTitle: 'Page Not Found',
  path: '/404'
  });
};
```

Figure 4.20: The HMI error page controller

Models

There was only one model developed for the HMI at this stage which serves the purpose of providing visualisation of data on the HMI home page. This model is called within the ‘controllers/hmi.js’, .getIndex() function which takes the object data of the ‘Device’ object allowing it to be used within the ‘index.ejs’ page.

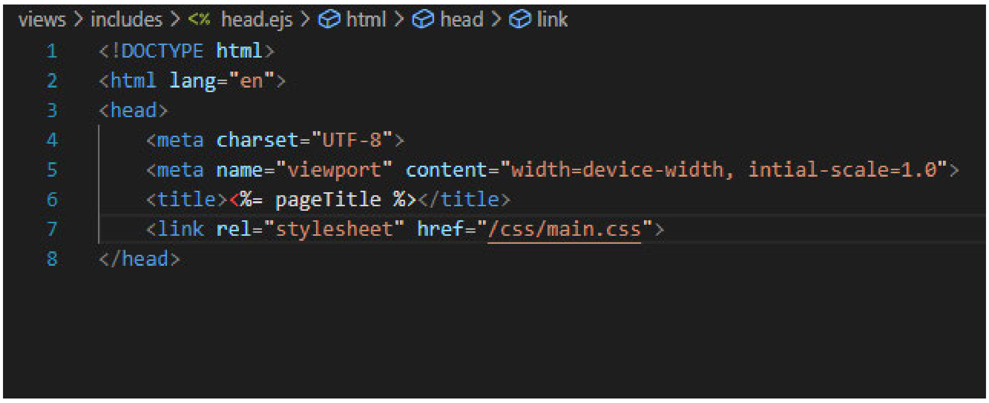


```
models > JS devices.js > ...
// The below describes a new model which is then exported

const Device =  {
    tag: 'PIT-210',
    value: '100',
    description: 'Pressure Transmitter'
};

module.exports = {
    Device
};

```

Figure 4.21: The HMI Device Model

Views

There are two display pages developed for the project HMI display, these were the homepage for testing, and an error / 404 page. To allow these to operate, EJS middleware was paired up with HTML, CSS and the existing JavaScript coded sections. The structure for the views section of the MVC system follows a top-down approach whereby the pages are listed under the ‘views’ folder, followed by the EJS ‘includes’ folder as seen in Figure 4.21. The ‘includes’ folder holds EJS link files which can be directly called in EJS page files allowing nesting of code sections, simplifying the development and readability of the code.



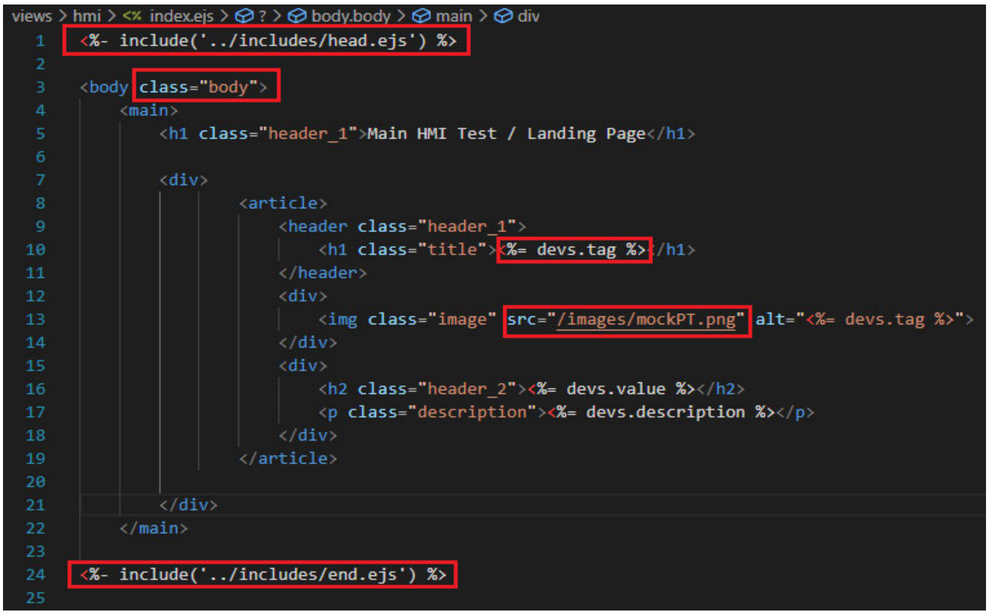
```
views > includes > <% head.ejs > html > head > link
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, intial-scale=1.0">
    <title><%= pageTitle %></title>
    <link rel="stylesheet" href="/css/main.css">
</head>
```

Figure 4.22: EJS includes ‘Head’ file

The ‘public’ folder in Figure 4.21 is linked to the MVC system within the ‘server.js’ file by importing the path module then using path.join in the below command (Express. 2024):

‘app.use(express.static(path.join(__dirname, ’/public’)));’

This links the sub-folder content such as CSS styling files and images for use when called appropriately in the MVC system. Once the controller has been called it attempts to serve the request by rendering the listed page. In the homepage case this is the ‘/’ page, which refers to the ‘http://localhost:3000/ ’ page being the index page. This then pulls the EJS file and renders it with the CSS stylesheet listed, source images and model data.



```
<%- include('../includes/head.ejs') %>

<body class="body">
    <main>
        <h1 class="header_1">Main HMI Test / Landing Page</h1>

        <div>
                <article>
                    <header class="header_1">
                        <h1 class="title"><%= devs.tag %></h1>
                    </header>
                    <div>
                        <img class="image" src="/images/mockPT.png" alt="<%= devs.tag %>">
                    </div>
                    <div>
                        <h2 class="header_2"><%= devs.value %></h2>
                        <p class="description"><%= devs.description %></p>
                    </div>
                </article>

        </div>
    </main>

<%- include('../includes/end.ejs') %>

```

Figure 4.23: HMI homepage ‘views/hmi/index.ejs’ file

The key elements of the HMI page are highlighted and listed below.

- ‘Includes’ commands - These allow EJS code sections to be embedded within the page code.
- ‘Class’ commands - Call style properties within the CSS file which is listed in the head.ejs file.
- EJS ‘%’ commands. These link external variables such as those listed by the ‘Device’ model.
- Image links. Because the public folder is defined using path.join, contents in the image folder can be referenced directly and called when the page loads.

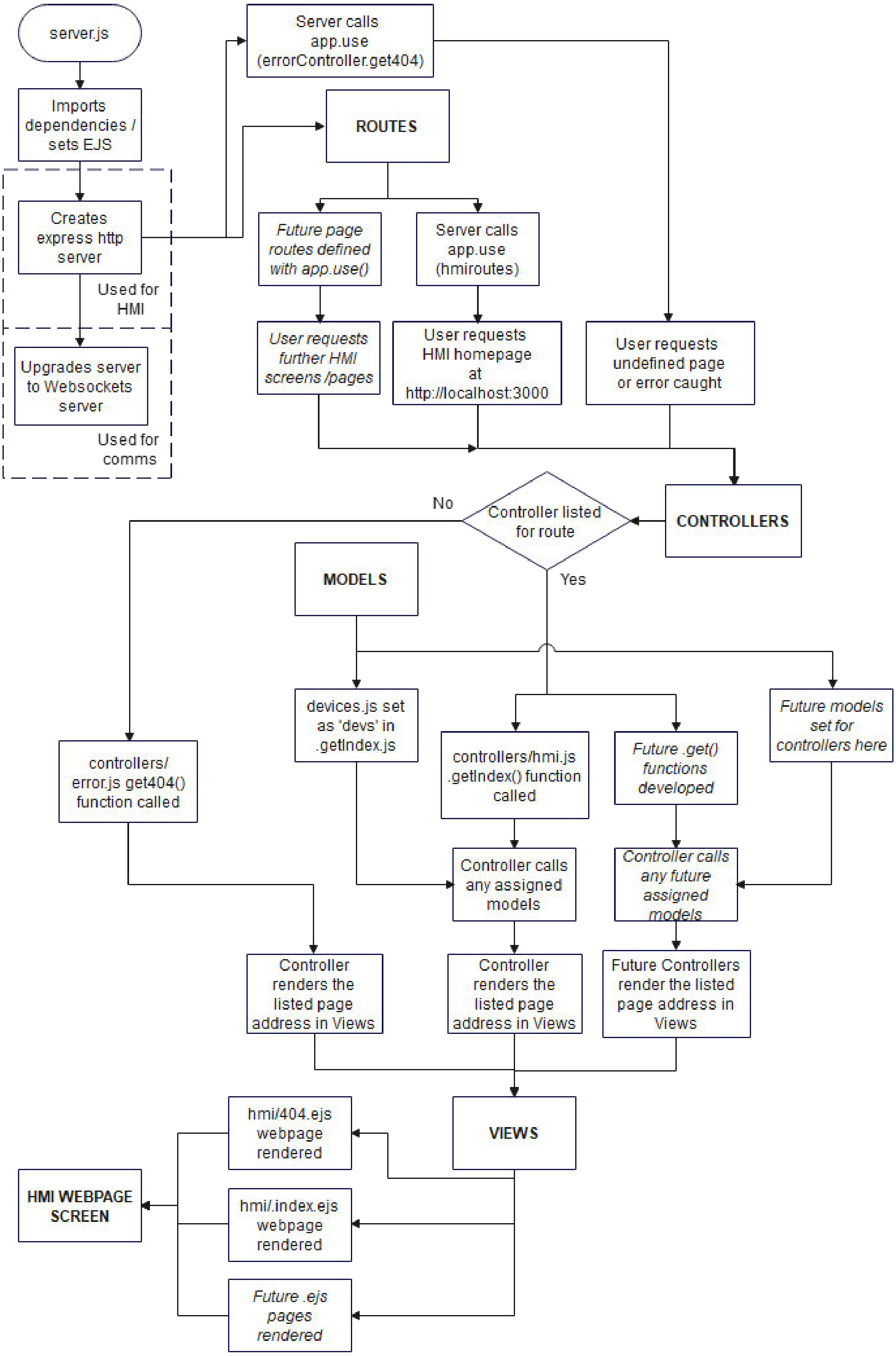

server.js
Imports dependencies / sets EJS
Creates express http server
Used for HMI
Upgrades server to Websockets server
Used for comms
Server calls app.use (errorController.get404)
ROUTES
Future page routes defined with app.use()
Server calls app.use (hmiroutes)
User requests further HMI screens /pages
User requests HMI homepage at http://localhost:3000
User requests undefined page or error caught
CONTROLLERS
Controller listed for route
No
Yes
MODELS
devices.js set as 'devs' in .getIndex.js
controllers/ error.js get404() function called
controllers/hmi.js .getIndex() function called
Future .get() functions developed
Future models set for controllers here
Controller calls any assigned models
Controller calls any future assigned models
Controller renders the listed page address in Views
Controller renders the listed page address in Views
Future Controllers render the listed page address in Views
VIEWS
hmi/404.ejs webpage rendered
hmi/.index.ejs webpage rendered
Future .ejs pages rendered
HMI WEBPAGE SCREEN

Figure 4.24: HMI Model-View-Controller Software Flowchart.

4.3.7 System Information and Data Monitoring

Distributed Control Systems require close monitoring of hardware and software resources to ensure reliability and performance can be maintained. To create a system capable of operating and enabling such system monitoring the NodeJS library packages 'OS' and 'process' were installed and software routines were developed.

createSysMap()

This function was a base level function used directly within server.js and also passed into the sysInfo() function as a parameter which was then subsequently called within the asynchronous setInterval function sendSystemData(). The createSysMap() function uses in-built NodeJS features to pull the operating system memory usage data which are assigned to three variables. These variables are then placed in a system data map and converted to a JSON format ready for WebSockets transmission, then returned as the function output.

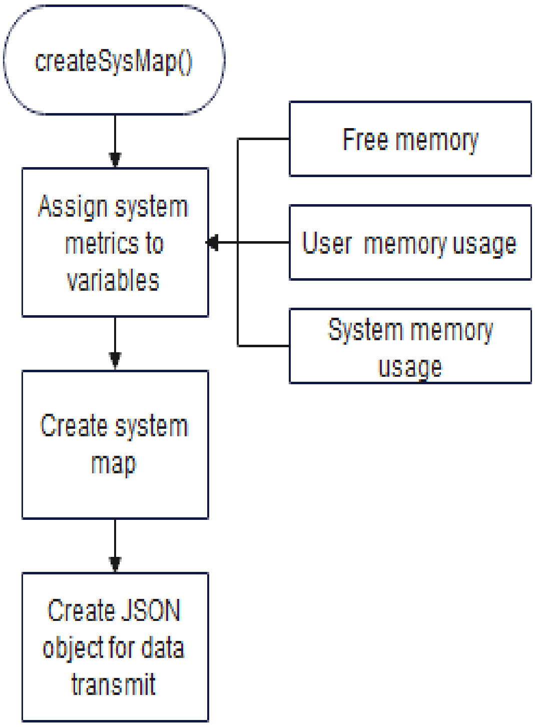


Figure 4.25: The 'createSysMap()' function

sendSystemData() and sysInfo()

The sysInfo() function was used by the two clients within the sendSystemData() function. The latter function called the sysInfo() function within a setInterval periodically executed cycle, allowing system data to be sent to the server.

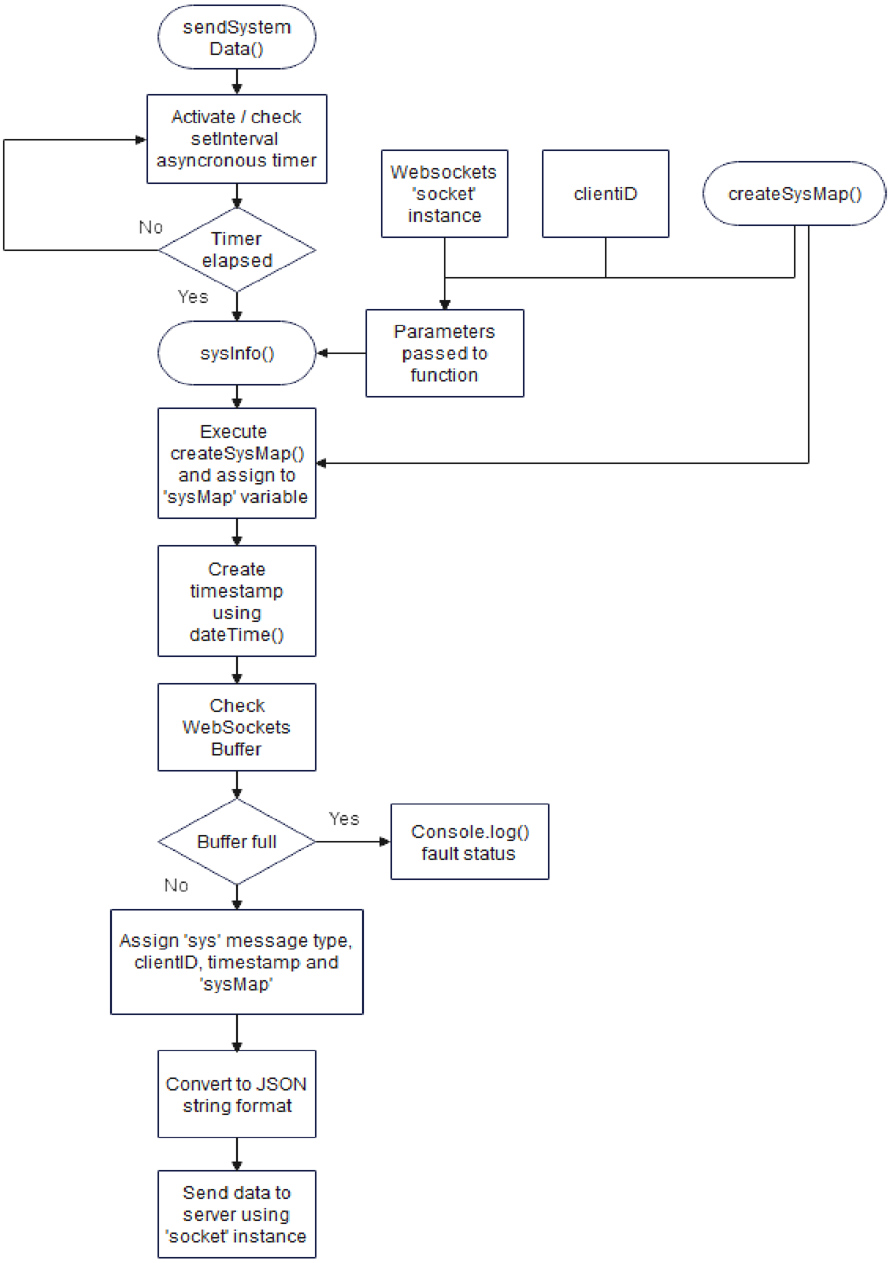

sendSystem Data()
Activate / check setInterval asyncronous timer
Timer elapsed
No
Yes
Websockets 'socket' instance
clientiD
createSysMap()
Parameters passed to function
sysInfo()
Execute createSysMap() and assign to 'sysMap' variable
Create timestamp using dateTime()
Check WebSockets Buffer
Buffer full
Yes
Console.log() fault status
No
Assign 'sys' message type, clientID, timestamp and 'sysMap'
Convert to JSON string format
Send data to server using 'socket' instance

Figure 4.26: The 'sysInfo()' function

4.4 Chapter Summary

This chapter has reviewed the hardware and software design processes chosen to develop a working distributed control system from embedded systems and web-based software systems. The key sections have developed a level of functionality as per the intended titles, the results of which are in the following section. The system as designed, exists as a structure capable of further enhancement in many of the selected software areas due to the use of classes and object-oriented design.

Chapter 5

Results and Discussion

5.1 Chapter Overview

This chapter reviews the test results delivered during development and operation of the distributed control system. The two major sections offer results from early stage development and final stage test results of the running system.

5.2 Preliminary Test Results

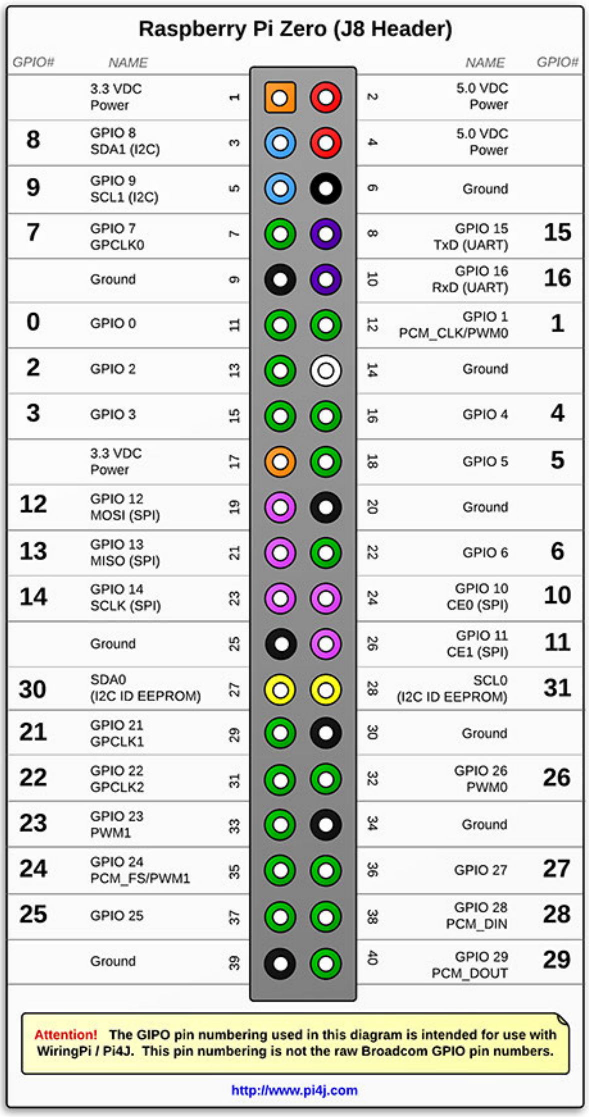
This section details the preliminary results to enable operation as desired with the project DCS. These are low-level tests which were critical to ensure further system functionality could be achieved later as the system was developed.

5.2.1 General Purpose Input Output Testing

Much of the system Input / Output testing was completed through simulated data to ensure a high number of values could be fed into the database and over the WebSockets protocol. Aside to this, enabling and testing of the input and output signals via the control system was seen as an important aspect. Once the GPIO configuration was enabled via the Pi Zero W preferences tab (detailed in Figure D.3 of Appendix D), the NodeJS applications could be set up.

A popular GPIO package for this is ‘rpio’. This was installed using ‘sudo apt install rpio’ within terminal, then imported within the client application function UpdateIO(). This allowed scanning of the GPIO status as input or setting as output pins. Performance wise the GPIO are a high-speed I/O driven by the Raspberry Pi clock capable of switching above KHz values.

By connecting a simple circuit to the GPIO 7 and Ground pins shown in Figure 5.1 below, a test LED circuit was used to confirm signals could drive the outputs of the Pi Zero W.



Raspberry Pi Zero (J8 Header)

GPIO#	NAME			NAME	GPIO#
	3.3 VDC Power	1	2	5.0 VDC Power	
8	GPIO 8 SDA1 (I2C)	3	4	5.0 VDC Power	
9	GPIO 9 SCL1 (I2C)	5	6	Ground	
7	GPIO 7 GPCLK0	7	8	GPIO 15 TxD (UART)	15
	Ground	9	10	GPIO 16 RxD (UART)	16
0	GPIO 0	11	12	GPIO 1 PCM_CLK/PWM0	1
2	GPIO 2	13	14	Ground	
3	GPIO 3	15	16	GPIO 4	4
	3.3 VDC Power	17	18	GPIO 5	5
12	GPIO 12 MOSI (SPI)	19	20	Ground	
13	GPIO 13 MISO (SPI)	21	22	GPIO 6	6
14	GPIO 14 SCLK (SPI)	23	24	GPIO 10 CE0 (SPI)	10
	Ground	25	26	GPIO 11 CE1 (SPI)	11
30	SDA0 (I2C ID EEPROM)	27	28	SCL0 (I2C ID EEPROM)	31
21	GPIO 21 GPCLK1	29	30	Ground	
22	GPIO 22 GPCLK2	31	32	GPIO 26 PWM0	26
23	GPIO 23 PWM1	33	34	Ground	
24	GPIO 24 PCM_FS/PWM1	35	36	GPIO 27	27
25	GPIO 25	37	38	GPIO 28 PCM_DIN	28
	Ground	39	40	GPIO 29 PCM_DOUT	29

Attention! The GIPO pin numbering used in this diagram is intended for use with WiringPi / Pi4J. This pin numbering is not the raw Broadcom GPIO pin numbers.

http://www.pi4j.com

Figure 5.1: Raspberry Pi Zero W GPIO Header Layout (Pi4J. 2019)

The below circuit and code was used to enable the output, confirming the Pi's GPIO capability. Similar input values were read using a switch on pins 7 & 9 proving discrete I/O control and monitoring to the client units was capable of integrating into the applications.

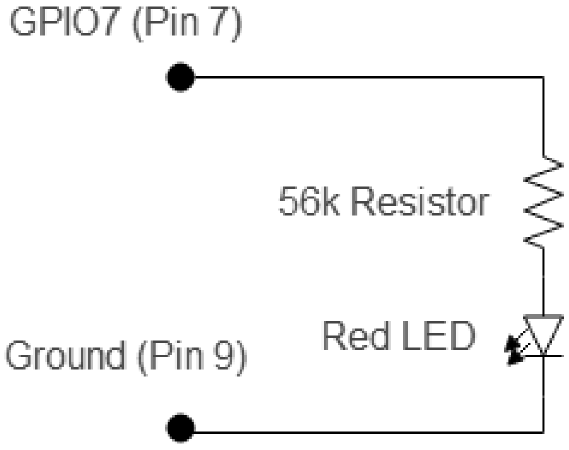
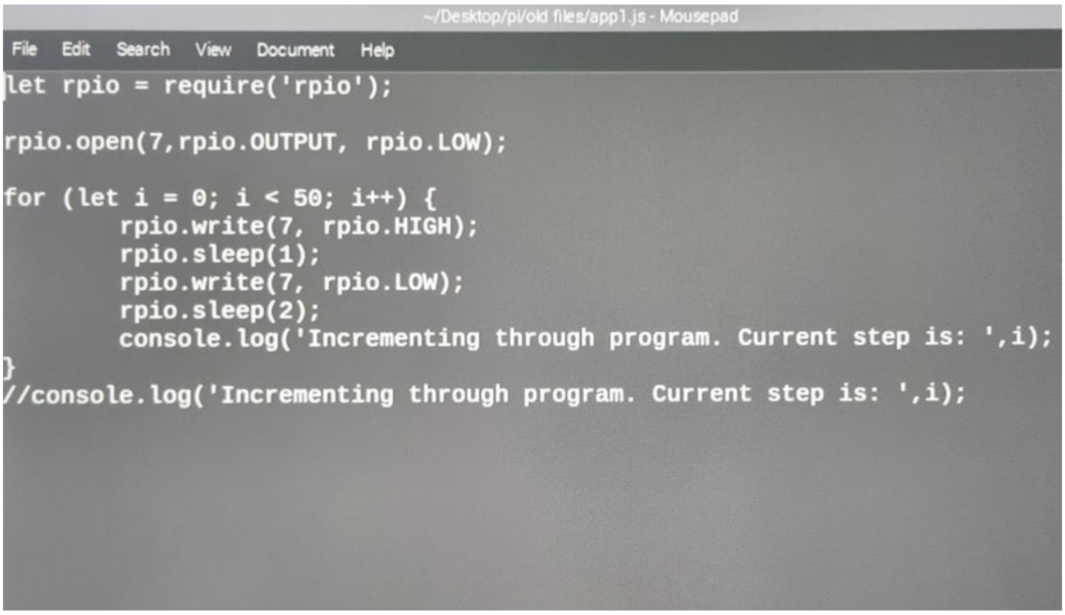


Figure 5.2: Pi Zero W GPIO test circuit.


~/Desktop/pi/old files/app1.js - Mousepad
File Edit Search View Document Help

```
let rpio = require('rpio');

rpio.open(7,rpio.OUTPUT, rpio.LOW);

for (let i = 0; i < 50; i++) {
        rpio.write(7, rpio.HIGH);
        rpio.sleep(1);
        rpio.write(7, rpio.LOW);
        rpio.sleep(2);
        console.log('Incrementing through program. Current step is: ',i);
}
//console.log('Incrementing through program. Current step is: ',i);
```

Figure 5.3: NodeJS GPIO test code.

5.2.2 Ping Testing of Ethernet Connections

To initially prove layer-2/3 communication was possible across the three embedded systems, testing was required. By using the downloaded net-tools package as listed in section D.3.1, connections could be proven on the network. The following steps were completed:

Connect hardware - Connect ethernet Cat6 cables, adaptors and Network Switch.

Assign Static IP Addresses - This step was shown in Figure D.2.

Confirm IP locked - Once set, refresh network and ensure IP is not reassigned by running the 'ifconfig'command.

Ping test all connections - Once IP's were set the ping command was used on all connections.

The MCU IP addresses were as per Figure 4.8, and were confirmed using the ping command 'ping 192.168.0.4' from the operating system to the target connection. Further to this, the latency was checked as a reference against future WebSockets connections with the below average values taken over 3 sets of 5 return results.

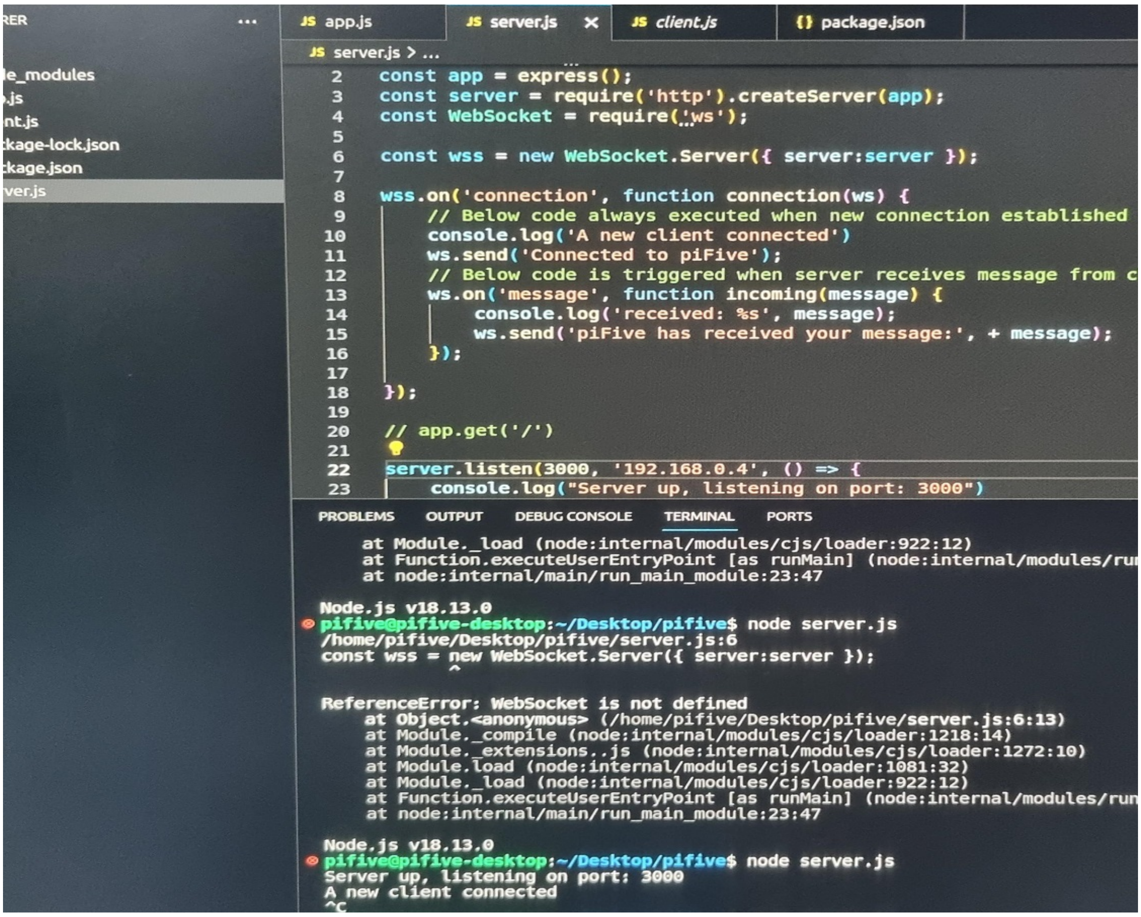
Description	Originating System IP	Description	Destination System IP	Time (ms)
Client 1	192.168.0.2	Server	192.168.0.4	2
Client 1	192.168.0.2	Client 2	192.168.0.3	4
Client 2	192.168.0.3	Server	192.168.0.4	2
Client 2	192.168.0.3	Client 1	192.168.0.2	4
Server	192.168.0.4	Client 1	192.168.0.2	1.5
Server	192.168.0.4	Client 2	192.168.0.3	1.5

Table 5.1: Ping Connection and Latency results across the network

5.2.3 Basic WebSockets Connection

After confirmation of the network ping testing, the basic NodeJS / Express application was initialised on all three MCU's. Then using the basic WebSockets implementation referenced in subsection 4.3.3, a WebSockets Server / Client structure was built. Issues were encountered in configuring the target IP addresses within the server and client initially, after which a connection was established, allowing a standard message to be sent and received at server and client ends.

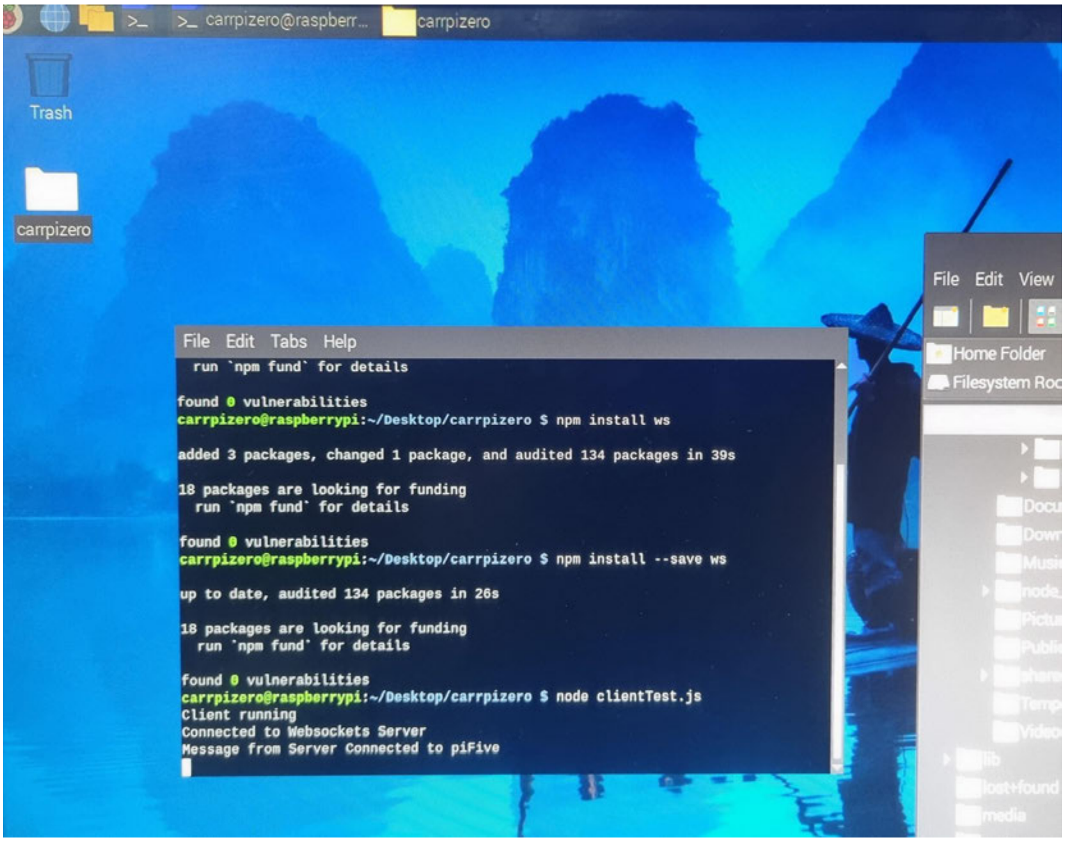
This was completed using the WebSockets 'send', 'connection' and 'on message' functions, available from the 'ws' library. This first-instance testing was completed within the main server and client application scripts prior to the development of separate script files and nested function calls. The results to confirm this testing was visual confirmation of string console output as seen in the reference Figures below.



```
const app = express();
const server = require('http').createServer(app);
const WebSocket = require('ws');

const wss = new WebSocket.Server({ server:server });

wss.on('connection', function connection(ws) {
    // Below code always executed when new connection established
    console.log('A new client connected')
    ws.send('Connected to piFive');
    // Below code is triggered when server receives message from c
    ws.on('message', function incoming(message) {
        console.log('received: %s', message);
        ws.send('piFive has received your message:', + message);
    });

});

// app.get('/')

server.listen(3000, '192.168.0.4', () => {
    console.log("Server up, listening on port: 3000")
```

```
    at Module._load (node:internal/modules/cjs/loader:922:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/ru
    at node:internal/main/run_main_module:23:47

Node.js v18.13.0
pifive@pifive-desktop:~/Desktop/pifive$ node server.js
/home/pifive/Desktop/pifive/server.js:6
const wss = new WebSocket.Server({ server:server });
            ^

ReferenceError: WebSocket is not defined
    at Object.<anonymous> (/home/pifive/Desktop/pifive/server.js:6:13)
    at Module._compile (node:internal/modules/cjs/loader:1218:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1272:10)
    at Module.load (node:internal/modules/cjs/loader:1081:32)
    at Module._load (node:internal/modules/cjs/loader:922:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run
    at node:internal/main/run_main_module:23:47

Node.js v18.13.0
pifive@pifive-desktop:~/Desktop/pifive$ node server.js
Server up, listening on port: 3000
A new client connected
^C
```

Figure 5.4: Pi Five WebSockets Server connectivity and message confirmation.



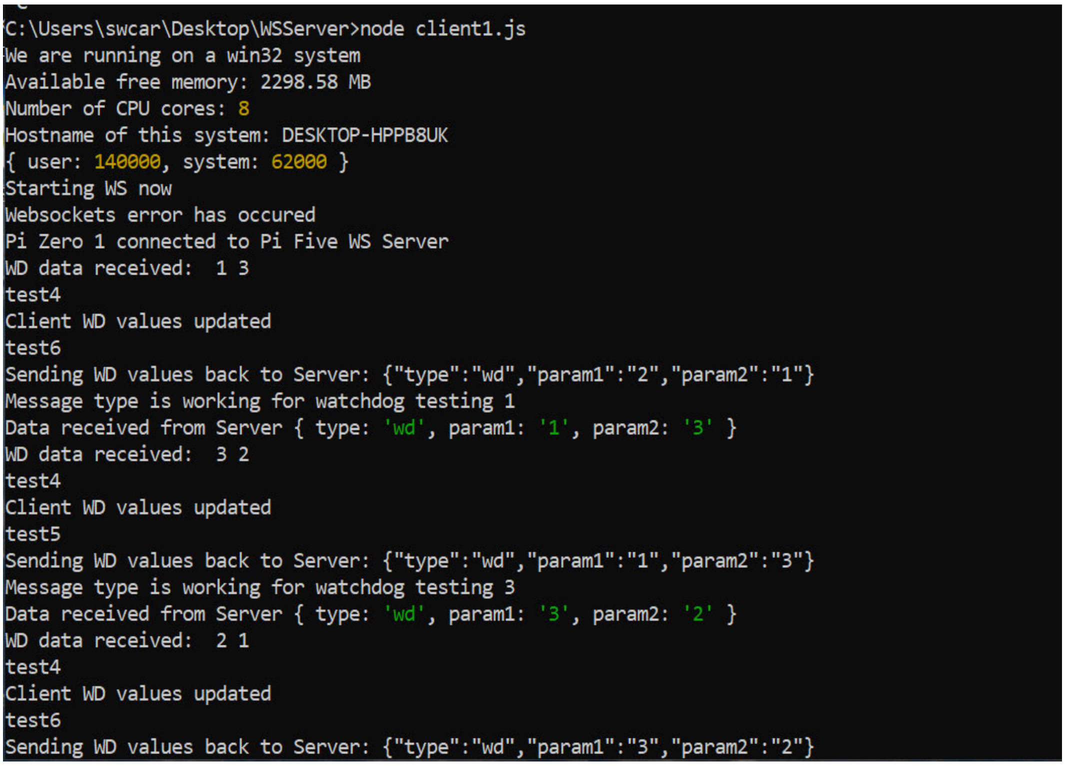
```
  run `npm fund` for details

found 0 vulnerabilities
carrpizero@raspberrypi:~/Desktop/carrpizero $ npm install ws

added 3 packages, changed 1 package, and audited 134 packages in 39s

18 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
carrpizero@raspberrypi:~/Desktop/carrpizero $ npm install --save ws

up to date, audited 134 packages in 26s

18 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
carrpizero@raspberrypi:~/Desktop/carrpizero $ node clientTest.js
Client running
Connected to Websockets Server
Message from Server Connected to piFive
```

Figure 5.5: Pi Zero W WebSockets Client connectivity and message confirmation.

5.2.4 Terminal and Console Output Logging

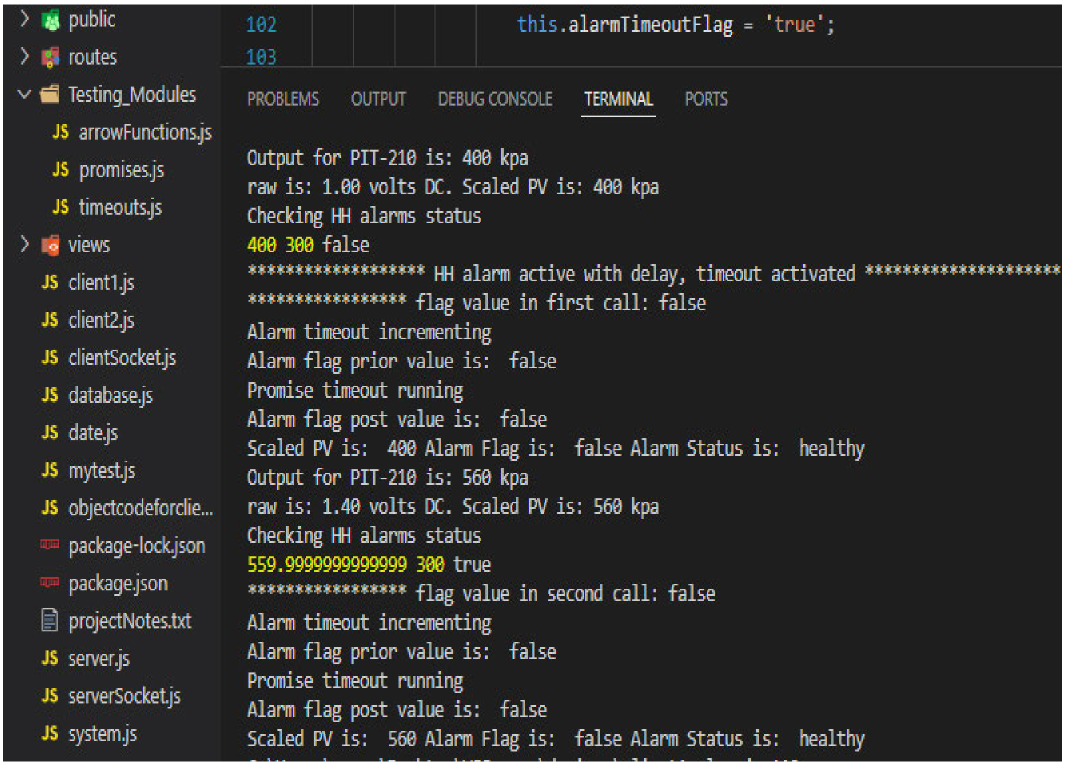
Once the base communication system and application was proven operational, the structure was formed into a separate folder and file system allowing improved, referenced code development. As the major system functions were developed a large amount of testing was carried out with the console log function used to confirm operation and sequence.

Critical aspects of operation centred on ensuring order of operations were maintained and that sections of code were confirmed functioning. By adding test points within the code the console log assisted to confirm these requirements. In other cases, custom messages were added with timestamps to visibly confirm program function. These confirmed results are used for reference in the following Detailed Test Results section 5.3. Examples are shown below in which the log displays point testing and custom timestamped messages within sequenced operations.



```
C:\Users\swcar\Desktop\WSServer>node client1.js
We are running on a win32 system
Available free memory: 2298.58 MB
Number of CPU cores: 8
Hostname of this system: DESKTOP-HPPB8UK
{ user: 140000, system: 62000 }
Starting WS now
Websockets error has occured
Pi Zero 1 connected to Pi Five WS Server
WD data received:  1 3
test4
Client WD values updated
test6
Sending WD values back to Server: {"type":"wd","param1":"2","param2":"1"}
Message type is working for watchdog testing 1
Data received from Server { type: 'wd', param1: '1', param2: '3' }
WD data received:  3 2
test4
Client WD values updated
test5
Sending WD values back to Server: {"type":"wd","param1":"1","param2":"3"}
Message type is working for watchdog testing 3
Data received from Server { type: 'wd', param1: '3', param2: '2' }
WD data received:  2 1
test4
Client WD values updated
test6
Sending WD values back to Server: {"type":"wd","param1":"3","param2":"2"}
```

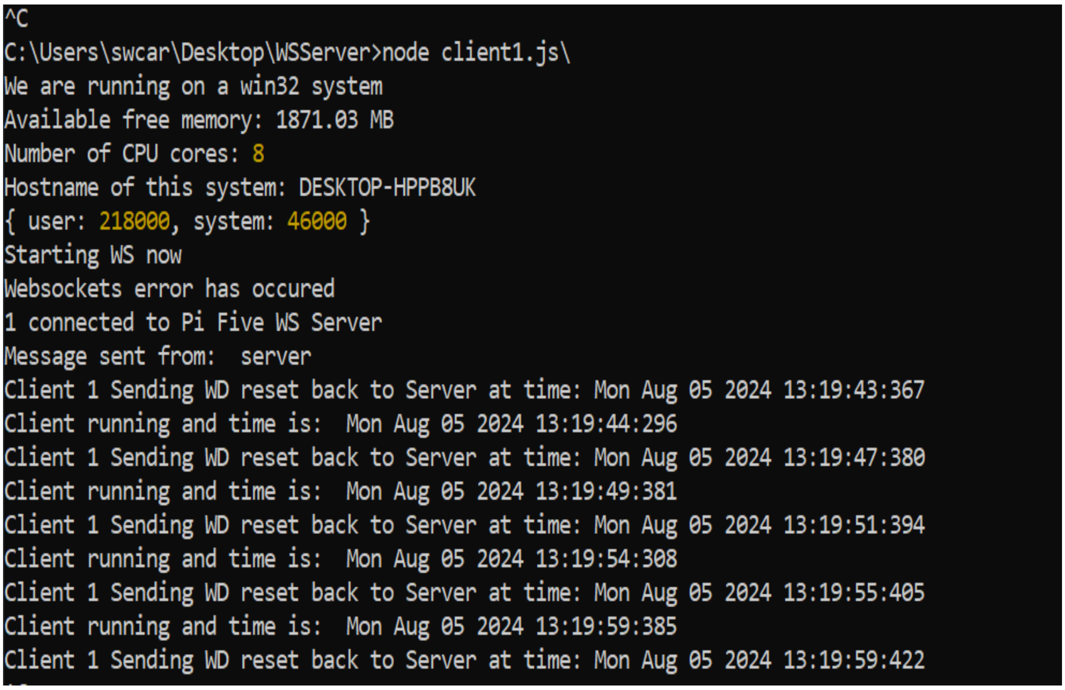
Figure 5.6: Code execution with added test points.



```
            this.alarmTimeoutFlag = 'true';
```

```
Output for PIT-210 is: 400 kpa
raw is: 1.00 volts DC. Scaled PV is: 400 kpa
Checking HH alarms status
400 300 false
******************* HH alarm active with delay, timeout activated *********************
***************** flag value in first call: false
Alarm timeout incrementing
Alarm flag prior value is:  false
Promise timeout running
Alarm flag post value is:  false
Scaled PV is:  400 Alarm Flag is:  false Alarm Status is:  healthy
Output for PIT-210 is: 560 kpa
raw is: 1.40 volts DC. Scaled PV is: 560 kpa
Checking HH alarms status
559.9999999999999 300 true
***************** flag value in second call: false
Alarm timeout incrementing
Alarm flag prior value is:  false
Promise timeout running
Alarm flag post value is:  false
Scaled PV is:  560 Alarm Flag is:  false Alarm Status is:  healthy
```

Figure 5.7: Promise testing with custom message confirmations of sequence.



```
^C
C:\Users\swcar\Desktop\WSServer>node client1.js\
We are running on a win32 system
Available free memory: 1871.03 MB
Number of CPU cores: 8
Hostname of this system: DESKTOP-HPPB8UK
{ user: 218000, system: 46000 }
Starting WS now
Websockets error has occured
1 connected to Pi Five WS Server
Message sent from:  server
Client 1 Sending WD reset back to Server at time: Mon Aug 05 2024 13:19:43:367
Client running and time is:  Mon Aug 05 2024 13:19:44:296
Client 1 Sending WD reset back to Server at time: Mon Aug 05 2024 13:19:47:380
Client running and time is:  Mon Aug 05 2024 13:19:49:381
Client 1 Sending WD reset back to Server at time: Mon Aug 05 2024 13:19:51:394
Client running and time is:  Mon Aug 05 2024 13:19:54:308
Client 1 Sending WD reset back to Server at time: Mon Aug 05 2024 13:19:55:405
Client running and time is:  Mon Aug 05 2024 13:19:59:385
Client 1 Sending WD reset back to Server at time: Mon Aug 05 2024 13:19:59:422
```

Figure 5.8: Client 1 watchdog timestamp and custom message confirmations.

5.3 Detailed Test Results

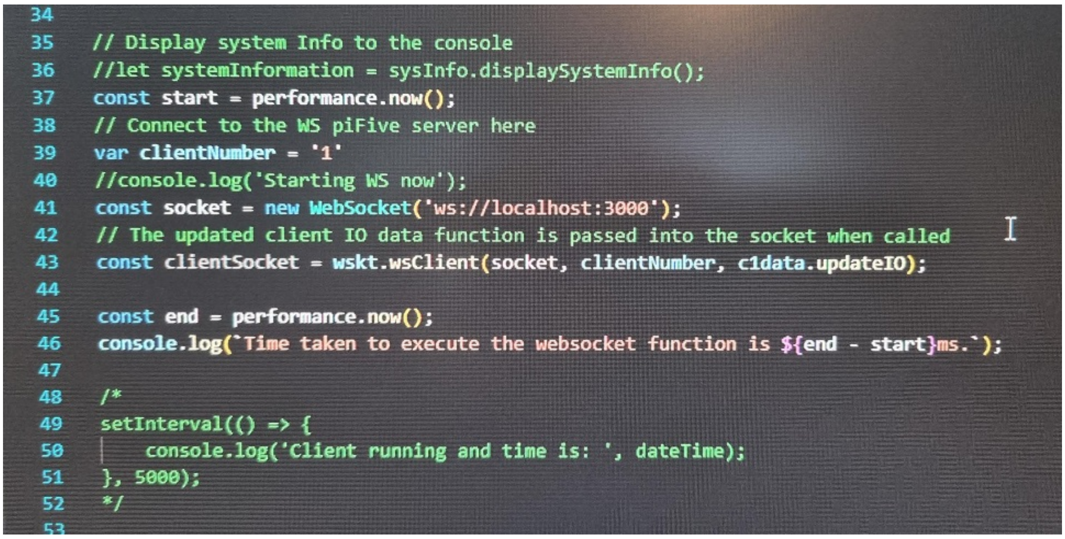
For a reliable, industrial-style DCS to be achieved, system integrity, combined with operation and reliability had to be proven during the development of this project. The below test sections focus on specific system operability, required to reflect such industrial level functions.

- Time Logging of Data & Execution Timekeeping
- Data Management and Storage
- Communications Reliability
- Human-Machine Interface Development Results
- Sequential and Asynchronous Event Operation
- Analog / Pulse-Width Modulation - Input / Output Testing

The results from each of these areas have been encouraging with system operation proven and key application and communication functions realised. The communications network was able to constantly deliver data at highly efficient rates for each session and store the data into separate database tables for accessing. System data was also able to be extracted and stored, with further monitoring possible in future which would enhance the reliability when heavily loaded. Custom modules for analog device scaling, process variable measurement and alarm activation were created. These successfully demonstrated that sequenced, time-dependent activation of alarms can be reliably configured. These results are demonstrated below.

5.3.1 Time Logging of Data & Execution Timekeeping

To confirm the performance of the system execution in the time domain another NodeJS library called ‘performance hooks’ was utilised. First, it was downloaded using ‘npm i perf_hooks’ into each parent folder on the MCU’s. Then it was imported into the required script files and wrapped around the functions to monitor execution times.



```
// Display system Info to the console
//let systemInformation = sysInfo.displaySystemInfo();
const start = performance.now();
// Connect to the WS piFive server here
var clientNumber = '1'
//console.log('Starting WS now');
const socket = new WebSocket('ws://localhost:3000');
// The updated client IO data function is passed into the socket when called
const clientSocket = wskt.wsClient(socket, clientNumber, c1data.updateIO);

const end = performance.now();
console.log(`Time taken to execute the websocket function is ${end - start}ms.`);

/*
setInterval(() => {
    console.log('Client running and time is: ', dateTime);
}, 5000);
*/
```

Figure 5.9: client1.js execution time monitoring.

This approach allowed visual display combined with logging of the major function execution times to determine the performance change over time and system loading of each developed function. The highest execution times were associated with the two clients communication functions, owing to their lower computational power and the overhead for calling nested functions. These results are given in the following Table 5.2

MCU System	Function / Script Description	Min Exec Time (ms)	Max Exec Time (ms)	Average Exec Time (ms)
Client 1	client1.js	1107.19	1014.8	1046.87
Client 1	wsClient()	17.21	18.51	18.26
Client 1	ioToDB()	0.3	1.55	0.6
Client 1	clientComms WatchdogUpdate()	5.21	19.18	9.51
Client 2	client2.js	849.35	949.39	893.92
Client 2	wsClient()	14.87	34.77	17.93
Client 2	ioToDB()	0.28	1.04	0.485
Client 2	clientComms WatchdogUpdate()	6.25	31.96	11.68
Server	server.js	1.9	2.69	2.204
Server	wsServer()	0.007	0.025	0.010
Server	myDB()	7.64	9.94	8.353

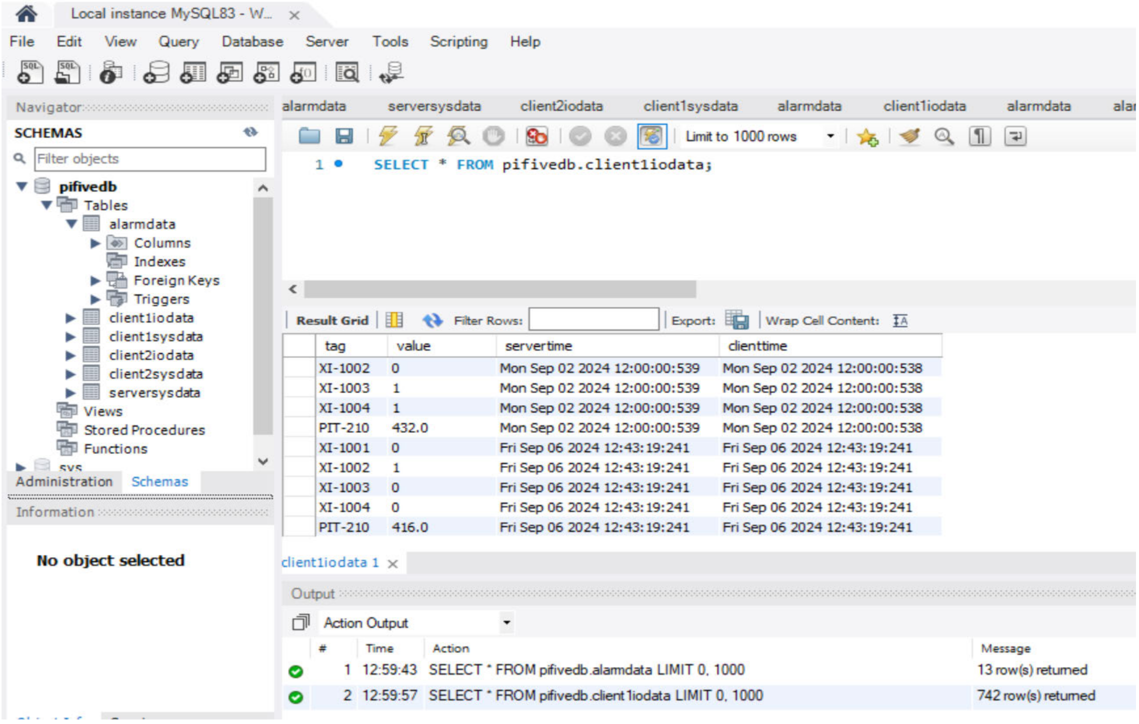
Table 5.2: Execution times in milliseconds for NodeJS programs and functions.

This data clearly demonstrated the high execution speeds possible for NodeJS based functions. The main client programs both had an approximate 1 second execution time, which is substantially longer than the other functions, however this was not an issue as these client programs call the cyclical inner functions into action and effectively only run once. Therefore, they do not have a significant effect on latency or performance once the two systems are running and communicating.

The main outcome / finding is that all programs once initialised achieve low execution times adding to system performance and assist with real-time updating of control and communication algorithms which is a key consideration for the embedded applications.

5.3.2 Data Management and Storage

The storage and access of data was seen as a key attribute for the project control system. The major points of focus were on creating segregated tables within the database and developing functions to allow these tables to be interacted with easily. The process of sending and receiving data, formatting it and then updating it into the MySQL database was achieved following a number of issues during. For the purpose of the project a total of six tables were created which captured client device values, alarms and system status data. The successful functioning of this database was a positive result within the project outcomes.



```
1  SELECT * FROM pifivedb.client1iodata;
```

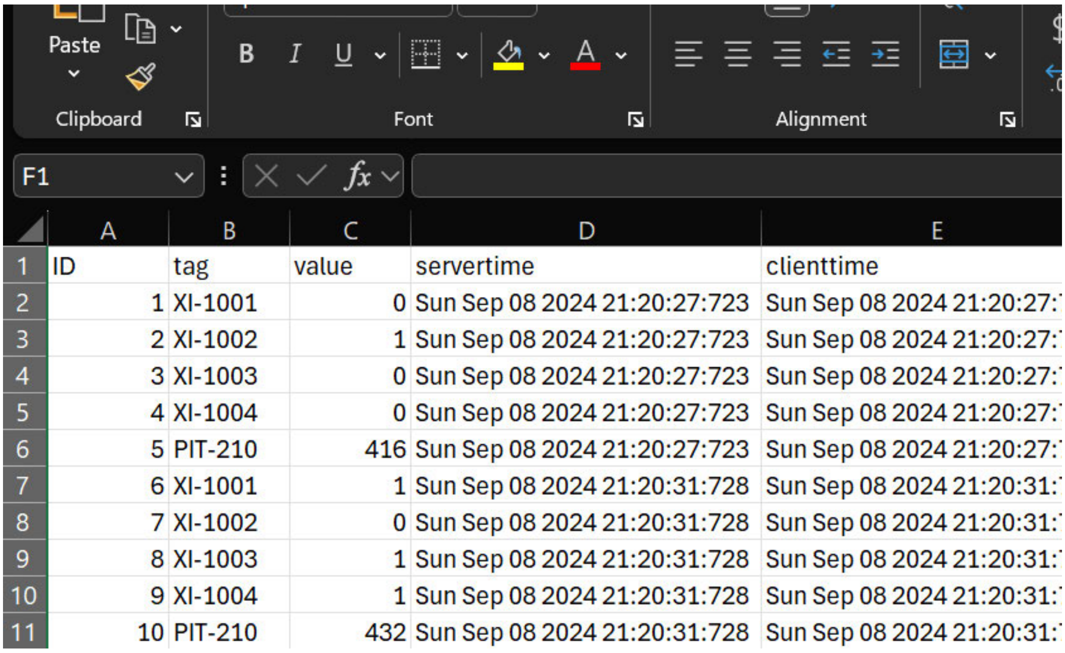
tag	value	servertime	clienttime
XI-1002	0	Mon Sep 02 2024 12:00:00:539	Mon Sep 02 2024 12:00:00:538
XI-1003	1	Mon Sep 02 2024 12:00:00:539	Mon Sep 02 2024 12:00:00:538
XI-1004	1	Mon Sep 02 2024 12:00:00:539	Mon Sep 02 2024 12:00:00:538
PIT-210	432.0	Mon Sep 02 2024 12:00:00:539	Mon Sep 02 2024 12:00:00:538
XI-1001	0	Fri Sep 06 2024 12:43:19:241	Fri Sep 06 2024 12:43:19:241
XI-1002	1	Fri Sep 06 2024 12:43:19:241	Fri Sep 06 2024 12:43:19:241
XI-1003	0	Fri Sep 06 2024 12:43:19:241	Fri Sep 06 2024 12:43:19:241
XI-1004	0	Fri Sep 06 2024 12:43:19:241	Fri Sep 06 2024 12:43:19:241
PIT-210	416.0	Fri Sep 06 2024 12:43:19:241	Fri Sep 06 2024 12:43:19:241

	#	Time	Action	Message
✓	1	12:59:43	SELECT * FROM pifivedb.alarmdata LIMIT 0, 1000	13 row(s) returned
✓	2	12:59:57	SELECT * FROM pifivedb.client1iodata LIMIT 0, 1000	742 row(s) returned

Figure 5.10: MySQL Workbench Visual data representation.

Another valuable outcome was the ability to interact with the database via an external connection which was achieved with the use of a laptop running MySQL Workbench 8.0 connected via the 5-Port network switch. The installation and use of Workbench 8.0 is covered in section D.3.

The ability to connect using this software effectively allows an external computer to access the data, visually review the tables, and extract the data in CSV format for external use and review. This allows periodic data to be accumulated for performance, reliability and monitoring purposes. The creation of custom tables allows highly specific data to be captured which makes the system adaptable for various system designs.



	A	B	C	D	E
1	ID	tag	value	servertime	clienttime
2	1	XI-1001	0	Sun Sep 08 2024 21:20:27:723	Sun Sep 08 2024 21:20:27:
3	2	XI-1002	1	Sun Sep 08 2024 21:20:27:723	Sun Sep 08 2024 21:20:27:
4	3	XI-1003	0	Sun Sep 08 2024 21:20:27:723	Sun Sep 08 2024 21:20:27:
5	4	XI-1004	0	Sun Sep 08 2024 21:20:27:723	Sun Sep 08 2024 21:20:27:
6	5	PIT-210	416	Sun Sep 08 2024 21:20:27:723	Sun Sep 08 2024 21:20:27:
7	6	XI-1001	1	Sun Sep 08 2024 21:20:31:728	Sun Sep 08 2024 21:20:31:
8	7	XI-1002	0	Sun Sep 08 2024 21:20:31:728	Sun Sep 08 2024 21:20:31:
9	8	XI-1003	1	Sun Sep 08 2024 21:20:31:728	Sun Sep 08 2024 21:20:31:
10	9	XI-1004	1	Sun Sep 08 2024 21:20:31:728	Sun Sep 08 2024 21:20:31:
11	10	PIT-210	432	Sun Sep 08 2024 21:20:31:728	Sun Sep 08 2024 21:20:31:

Figure 5.11: Database table data exported in CSV format.

The HMI is also able to query the database via the MVC framework which allows the real-time data to be embedded into the HMI display for device values. This was successfully demonstrated with updated device values from the database being displayed every second on the screen.

However, whilst this function was achieved it was implemented using a page refresh method which is not highly efficient. This is an area which could be improved given more time resources to further streamline and enhance the HMI element of the overall project.

5.3.3 Communications Reliability

With timestamped data at source and destination, combined with database recording, the system was able to provide valuable data to indicate the latency and connection reliability for the 3 system network. When reviewing the database timestamp data, it is important to note the distributed systems were not universally time synchronised. With the two timestamps, the relative offset was able to be monitored for each session duration that the server / clients were running. This provided insight as to the fluctuation and variability of the latency over time. Once operating with the 3 applications running, there were no communication failures encountered allowing extended data collection without issue.

Numerous individual tests were performed with three notable examples chosen to demonstrate the reliability of the application function and communication network combined. For the below tests, 20 simulated devices were created within each client, consisting of 16 discrete and 4 analog device objects to represent a realistic IO system assignment. These had simulated values scanned into the objects, written to the datamaps and sent across the network. Upon being received by the server they were formatted and written into the according database tables. Many individual tests of this nature were performed while the system was developed, once proven the below tightly controlled tests were performed with highly positive results.

The below three tests were performed, recorded and exported using WorkBench 8.0 without a single error during operation. Each test increased the speed of data transmit and reduced the watchdog interval allowing faster error detection and improving reliability.

1. 1.5 hour test - 4 second watchdog interval, 6 second watchdog timeout, 500 millisecond device updates to server (Total 40 devices, more than 420,000 combined database data entries without error).

2. 21 hour test - 400 millisecond watchdog interval, 500 millisecond watchdog timeout, 2 second device updates (40 devices, more than 1.5 million database data entries without error)

3. 12 hour test - 400 millisecond watchdog interval, 500 millisecond watchdog timeout, 500 millisecond device updates (40 devices, more than 1.9 million database data entries without error)

When reviewing the CSV data exports, the below findings were taken.

- Maximum change in timestamp offset (drift) over any session: 7 milliseconds
- Minimum change in timestamp offset (drift) over any session: 2 milliseconds
- Average offset between systems when online: +/- 4 milliseconds
  (This is latency and error combined between system times - due to no synchronisation)
- Minimum Data poll rate interval Client to Server achieved: 500 milliseconds
- Maximum loaded session duration completed: 21 hours (with 40 simulated devices)

These results, when reviewed against the Ping Testing of Ethernet Connections subsection 5.2.2 results show that even with a 2-4ms latency on the level-3 layer the WebSockets protocol can deliver data efficiently. The results also indicate that with the current system design the applications and communications can reliably maintain fast data exchange for control applications.

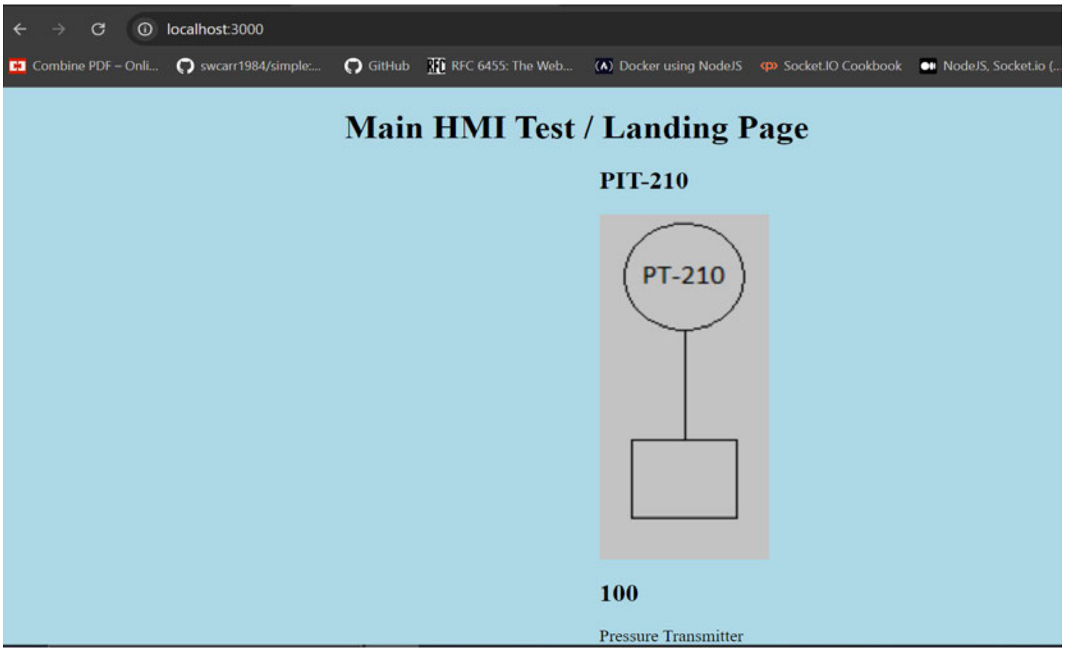
By separating certain elements of the communications and data streams, select behaviour was developed for each sectioned grouping. While the alarm development within the analog device module / class successfully created event-driven alarms, for the purpose of communication, these alarms were cyclically polled via the WebSockets protocol. It was possible to set these alarms as event-driven within the communication scheme also, however due to the extremely low latency and ability to poll at a high rate this was not opted for. If network latency increased, the ability is provided to activate a WebSockets alarm message prior to a standard polling regime, speeding the alarm response rate.

Another key outcome was realised whereby alarms which only need to be scanned or recognised when activated, could be set as event-driven elements within a defined grouping. This was proven when the one-shot messaging function to the database was configured, which ensured alarm-flooding did not occur. Customisable conditions such as these, allowed tailored alarm response and storage into database tables.

5.3.4 Human-Machine Interface Development Results

The system HMI was able to be successfully rendered using the MVC approach stated in Sub-Section 4.3.6. While the display was not extensively developed in format and functional terms, the modular design permitted future development with custom graphics able to be created and embedded.

Heavily structured formatting of the page was not achieved due to other project demands, however the creation of individual device models with internal real-time data which was requested direct from the database was completed. This allowed the data to be rendered onto the display which in future could reflect a more representative, functional and dynamic set of graphical element.



Main HMI Test / Landing Page

PIT-210

100

Pressure Transmitter

Figure 5.12: Basic HMI home screen rendered using MVC framework.

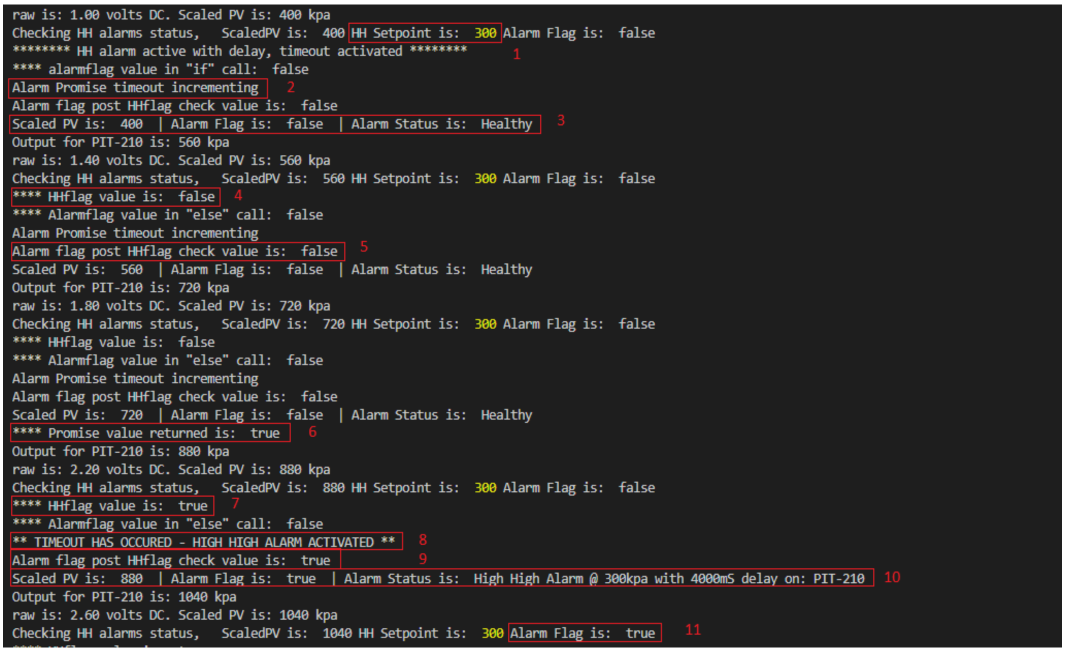
Further work could see such a page resemble a working process plant with interactive elements, high-performance HMI techniques employed and the ability to represent a multitude of control systems. While these are advanced functions, they can be built from the existing model structure with images, live-values and styling already proven within the low-level HMI structure delivered.

5.3.5 Sequential and Asynchronous Event Operation

One of the key themes within the project was the development of software with combined temporal and event-driven behaviour. The aim of the project was to deeply investigate ways in which both time-dependent and event-driven operations could be combined and assist in a functional distributed control architecture.

The analog device class aiDevice{} was developed with multiple internal methods which are documented in Sub-Section 4.3.4. This class, when invoked as separate object instances within the updateIO() function, successfully demonstrated the ability of NodeJS to operate in both temporal and event-driven modes.

The updateIO() function was periodically polled via the sendData() function held within clientSocket(). It used in-built, asynchronous actions common to NodeJS to achieve this. The system device polling functionality was activated by creating a worker thread external to the main NodeJS runtime which actioned a callback every instance the thread timer was elapsed. This process allowed development of sequenced periodic operations such as watchdog timers, device status and system data polling from client to server.



```
raw is: 1.00 volts DC. Scaled PV is: 400 kpa
Checking HH alarms status,   ScaledPV is:  400 HH Setpoint is:  300 Alarm Flag is:  false
******** HH alarm active with delay, timeout activated ********
**** alarmflag value in "if" call:  false
Alarm Promise timeout incrementing
Alarm flag post HHflag check value is:  false
Scaled PV is:  400  | Alarm Flag is:  false  | Alarm Status is:  Healthy
Output for PIT-210 is: 560 kpa
raw is: 1.40 volts DC. Scaled PV is: 560 kpa
Checking HH alarms status,   ScaledPV is:  560 HH Setpoint is:  300 Alarm Flag is:  false
**** HHflag value is:  false
**** Alarmflag value in "else" call:  false
Alarm Promise timeout incrementing
Alarm flag post HHflag check value is:  false
Scaled PV is:  560  | Alarm Flag is:  false  | Alarm Status is:  Healthy
Output for PIT-210 is: 720 kpa
raw is: 1.80 volts DC. Scaled PV is: 720 kpa
Checking HH alarms status,   ScaledPV is:  720 HH Setpoint is:  300 Alarm Flag is:  false
**** HHflag value is:  false
**** Alarmflag value in "else" call:  false
Alarm Promise timeout incrementing
Alarm flag post HHflag check value is:  false
Scaled PV is:  720  | Alarm Flag is:  false  | Alarm Status is:  Healthy
**** Promise value returned is:  true
Output for PIT-210 is: 880 kpa
raw is: 2.20 volts DC. Scaled PV is: 880 kpa
Checking HH alarms status,   ScaledPV is:  880 HH Setpoint is:  300 Alarm Flag is:  false
**** HHflag value is:  true
**** Alarmflag value in "else" call:  false
** TIMEOUT HAS OCCURED - HIGH HIGH ALARM ACTIVATED **
Alarm flag post HHflag check value is:  true
Scaled PV is:  880  | Alarm Flag is:  true  | Alarm Status is:  High High Alarm @ 300kpa with 4000mS delay on: PIT-210
Output for PIT-210 is: 1040 kpa
raw is: 2.60 volts DC. Scaled PV is: 1040 kpa
Checking HH alarms status,   ScaledPV is:  1040 HH Setpoint is:  300 Alarm Flag is:  true
```

Figure 5.13: High High delay on analog device using promise callbacks.

Further to this, the addition of promise-based callbacks extended the capability, with delay timers nested within the periodically polling function capable of separate event-driven responses for the alarm setpoints when they were activated. By creating separate system operations of this nature the event responses could be individually managed external to the ongoing polling functionality. This was an important result which successfully demonstrated the flexible nature of the combined approach to software development.

In reference to the numbered points displayed in Figure 5.13, the below test steps detail the order of synchronous and asynchronous operations.

1. An instance of an analog device is created, this simulated object provides code to increment the scaled variable each cycle. As the devices are scanned the checkAlarms() method compares all alarm setpoints to the scaled process variable. In this case the setpoint for a High High has been exceeded.
2. As the setpoint is activated and has a delay programmed, a promise-based timeout is executed starting the timer incrementing in an externally operated thread, separate to the cyclical checkAlarm() execution.
3. This point displays the healthy status as the timer has not elapsed and no callback has been returned from the worker pool.
4. The High High flag is therefore still set to false.
5. The overall alarm status flag is also still false.
6. After further cycles performing the same checks the asynchronous promise elapses, triggering the main thread callback which returns ’true’ activating the local variable.
7. During this device scan the variable is now read as ’true’ and the checkAlarms() method assigns a ’true’ status to the ’High High flag’.
8. A local console display also indicates the Timeout has occured purely for logging.
9. The overall alarm status test point executing post the High High assignment is now set to ’true’, returned from the checkAlarm() method which is sent with a message using the alarm WebSockets function and stored in the Database alarm table.
10. The alarm flag status is assigned along with the the current scaled process variable and the pre-programmed alarm logging status which automatically generates. This is sent to the database to store as below.

11. The main alarm flag is now set high within checkAlarms() for the specific device object. This is returned from the checkAlarm() method which is subsequently sent within a message using the alarm WebSockets function and stored in the Database alarm table.

5.3.6 Analog & Pulse-Width Modulation - Input / Output Testing

The distributed control system developed within this project held the ability to be easily installed and run on other embedded architectures which were capable of running a NodeJS framework with Ethernet network capabilities. However, as the Raspberry Pi series was utilised the element of hardware and software interfacing was investigated to determine the capabilities within the limited project time allocated.

The ADC Pi analog card was successfully interfaced with a high-accuracy voltage calibration Portacal 1000 unit giving excellent results within a 0.5% error tolerance across it's range of 0-5VDC. A 5-step rising / falling test was completed 3 times with no greater error than 0.5%. The results are given below showing the accurate and repeatable analog to digital conversion achieved through the ADC Pi interfacing card. Further details relating to the testing are in Appendix D Sub-Section D.3.4.

Input Voltage	**Raw Read**	**Scaled Voltage Read**	**Error %**
0.0	46	0.0004	<0.5
1.0	25835	0.9963	<0.5
2.0	51635	1.9927	<0.5
3.0	77437	2.9888	<0.5
4.0	103216	3.9846	<0.5
5.0	129020	4.9808	<0.5
4.0	103215	3.9844	<0.5
3.0	77419	2.9885	<0.5
2.0	51618	1.9923	<0.5
1.0	25813	0.9960	<0.5
0.0	17	0.0000	<0.5

Table 5.3: Results of analog scan and scale testing

The Pulse Width Modulation capability of the Pi Zero W units was partially proven successful during testing. While the output PWM signal was effective and accurate from the Pi Zero W pins, the MOSFET output board listed in Sub-Section 3.6.5 required a higher output signal to accurately operate between the 0-5V range. This was confirmed after a True-RMS multimeter was used in conjunction with a Picoscope to analyse the voltage and waveform outputs.

The output measured direct at the Pi Zero W unit worked sufficiently in pulse-width and timing terms even though the Voltage Module output suffered from both incorrect MOSFET operation and minor repeatability issues. With further voltage interfacing a working 0-5VDC analog output could potentially be delivered, however the minimal project time and resources allocated, expected lead time for alternate hardware or time for developing improved custom circuitry further solutions for this aspect of the project were not explored.

The resulting output values are given below.

PWM Output (%)	**Measured Voltage (DC)**	**Error (ABS %)**
0	0.0015	0.03
25	3.5955	46.8
50	4.3985	37.97
75	4.9108	23.22
100	4.9932	0.14
75	4.9287	23.57
50	4.4082	38.16
25	3.6042	47.08
0	0.0012	0.01

Table 5.4: Results of Pi Zero W PWM output testing

5.4 Chapter Summary

This chapter has documented the various outcomes that have been assessed through testing of the project distributed control system. It has shown how NodeJS runtime applications, in combination with embedded systems, can effectively deliver reliable and efficient communications, functional aspects and visual displays.

These results have been positive helping satisfy the project outcomes and have the potential with further research and development to deliver highly functional and reliable industrial level performance.

Chapter 6

Conclusions and Further Work

6.1 Chapter Overview

This chapter presents the conclusions that have been met in relation to the research results obtained, and design work conducted within the project. It reviews what has been achieved within the project, including delivery of the stated outcomes by satisfying the objectives specified.

6.2 Conclusions

From inception, this project aimed to investigate modern, Web-Based Technologies in order to determine if they could be used to develop effective distributed control system architectures. To confidently align with this aim, a number of outcomes were expected which would be used to ensure meaningful results and conclusions could be drawn from the research and design work undertaken. These outcomes are given below with the associated comments for each.

Outcome 1: Determine the suitability and benefits of a Full-Stack JavaScript based control system.

This was a broadly stated outcome to encompass the ability to create useful elements of a control system utilising a JavaScript-based, NodeJS stack. The following key results were delivered which confirmed the suitability of such an approach and showcased the benefits such as highly integrated code and operability.

- The Full-Stack system allowed seamless, single programming language data transfer between devices and database.
- Both time-dependent and event-driven techniques were successfully developed within the programs created.
- High-level programming libraries assisted to develop custom solutions.
- The HMI display was able to directly render from the application owing to the Full-Stack integration of the NodeJS, MVC framework developed.
- The MySQL database was able to operate using JSON formatted data to and from the server application.

Outcome 2: Report and display the performance-based results for WebSockets based communication, developed to meet the distributed architecture.

The development of a date and time-stamping function on server and client ends, in parallel with database recording of these stamped values was completed. These results successfully demonstrated that low millisecond transmission rates were consistently achieved over the network between applications.

The recording also allowed logging of any errors or connection closures during specific periods, of which there were no errors which caused disconnection.

Outcome 3: Clearly document the methods used to create and implement control system components and where the key benefits are gained in the process.

This outcome was delivered by completion of the Methodology Section 3 and the Functional Design Section 4, where the software process flow diagrams and associated descriptions detailed successful system component development.

Outcome 4: Show that low-cost hardware with open-source software can build up the layers of a performant control system.

Again, this outcome was delivered by completion of the Functional Design section 4, where the specified hardware and software elements were detailed. These elements combined, delivered a system capable of high-speed data transfer, control and database management.

Outcome 5: Detail how data can become more accessible and tailored with a full-stack approach.

The project saw a combination of a database and HMI display both being integrated directly into the server application. This assisted in showing how flexible and seamless the selection of a JavaScript-based full-stack solution was. With adaptable functions, specific network message types, alarm processing and remote access to the database possible, the developed DCS framework had highly-accessible data streams.

Outcome 6: Provide recommendations on further improvements and future developments using such technologies.

Given the applications were created with object-oriented construction methods which encapsulated internal operations, further modification and improvement is made easier. There are many aspects of an industrial DCS which are still to be developed with the project system, however, with the development of a working high-speed network, database interfaces, HMI display and device classes, further advances are much easier to accomplish. Recommendations on further specific work are given in Section 6.4.

6.3 Achievement of Research Objectives

To deliver the outcomes listed in the above Section 6.2, the research objectives below were all satisfied.

- Conduct a detailed review of the key components in mid-sized control systems and detail how the selected software can be utilised to develop both theoretical, and where possible, practical solutions of each equivalent section.
- Implement and test performance of a WebSockets based communication layer over multiple nodes to allow distributed data channels and control to be realised.
- Review and assess how an implementation of the NodeJS server runtime can be structured to meet potential schedule deadlines and deliver reliable core control system operation.
- Develop sound, re-useable, modular code and class-based systems for any control system architectures created.

6.4 Further Work

The project listed many of the major elements of a distributed control system with the aim to investigate, replicate functionality and assess. However, there are many further elements and functions which can extend this work using a similar process.

Options such as further developing interfacing hardware, classes for device management and highly specific control algorithms have not been completed within this scope. Each of these alone could take a great deal of time and resource to further enhance and test. The below list covers a number of future points which could continue this projects progress.

- Continue developing the WebSockets communication functions to include further message handling and data compression techniques as required.
- Add security implementations including WebSockets authentication and advanced monitoring features as the overall system is developed.
- Developing further class-based software for device interfacing and program control.
- Advancing the HMI using improved formatting with the live-data streams for process variables and optimised page element refresh techniques.
- Implementing class-based control algorithms to check real-time sensitive time-dependent operations such as PID controllers etc.
- Further testing hardware interfaces to allow control of field devices using the developed analog and digital classes.

6.5 Chapter Summary

This chapter has summarised the key research outcomes which have been delivered through satisfying the initially conceived project objectives. It has also provided a list of potential further work which can add value to the project aim and future implementation of such systems.

References

Amazon. (2024), '5 Pcs 3.3v/5v IRF520 MOSFET Driver Module', https://www.amazon.com.au/IRF520-MOSFET-Driver-Driving-Arduino/dp/B08F54VFSY?source=ps-sl-shoppingads-lpcontext&ref_=fplfs&psc=1&smid=ATKMJIEDSKG3R. [Online; accessed 5 June-2024].

Ancona, D., Franceschini, L., Delzanno, G., Leotta, M., Ribaudo, M. & Ricca, F. (2017), Towards Runtime Monitoring of Node.js and Its Application to the Internet of Things, *in* D. Pianini & G. Salvaneschi, eds, 'Proceedings First Workshop on Architectures, Languages and Paradigms for IoT, 2017', Vol. 264 of *EPTCS*, pp. 27–42. https://doi.org/10.4204/EPTCS.264.4.

Bellini, P., Cenni, D., Mitolo, N., Nesi, P., Pantaleo, G. & Soderi, M. (2022), 'High Level Control of Chemical Plant by Industry 4.0 Solutions', *Journal of Industrial Information Integration* **26**, 100276. https://www.sciencedirect.com/science/article/pii/S2452414X2100073X.

Bonfe', M., Fantuzzi, C. & Poretti, L. (2001), Plc object-oriented programming using iec61131-3 norm languages: An application to manufacture machinery, *in* '2001 European Control Conference (ECC)', pp. 3235–3240.

Branch, M. & Bradley, B. (2006), Real-time web-based system monitoring, *in* 'Conference Record of 2006 Annual Pulp and Paper Industry Technical Conference', pp. 1–4.

Budimir, M. (2018), 'What are IEC 61131-3 and PLCopen?" – Design World.', https://www.designworldonline.com/what-are-iec-61131-3-and-plcopen/. [Online; accessed 25 Feb-2024].

Core-Electronics. (2024), 'Raspberry Pi - Zero W', https://core-electronics.com.au/raspberry-pi-zero-w-wireless.html. [Online; accessed 6 May-2024].

ebay. (2024*a*), '0-5v to 4-20ma Linear Conversion Voltage to Current Transmitter Signal Module', https://www.ebay.com.au/itm/112419076150. [Online; accessed 5 June-2024].

ebay. (2024*b*), 'AC 240V To DC 24V Power Supply', https://www.ebay.com.au/itm/112843990043?chn=ps&_ul=AU&mkevt=1&mkcid=28&srsltid=AfmBOoq3rB9ZMf1HSU9FziEJHcuxDZmkhp_zU3CkVuWWIgcH7W7jpXpb5xM. [Online; accessed 8 June-2024].

Eernisse, M. (2020), 'EJS – Embedded Javascript Templates', https://ejs.co/. [Online; accessed 5 Mar-2024].

Express. (2024), 'Serving Static Files in Express', https://expressjs.com/en/starter/static-files.html. [Online; accessed 03 Aug-2024].

Expressjs. (2024), 'Express Middleware', https://expressjs.com/en/resources/middleware.html. [Online; accessed 5 Mar-2024].

Fang, Z. & Fu, Y. (2011), A networked embedded real-time controller for complex control systems, *in* '2011 Chinese Control and Decision Conference (CCDC)', pp. 3210–3215.

Fluke (2024), '4 Things You Need to Know About Industrial Ethernet', https://www.fluke.com/en-us/learn/blog/electrical/industrial-ethernet. [Online; accessed 27 Feb-2024].

Gillis, A. S. (2023), 'What is a DCS? – Tech Target', https://www.techtarget.com/whatis/definition/distributed-control-system. [Online; accessed 20 Feb-2024].

Gundall, M. & Schotten, H. D. (2021), 'Assessing Open Interfaces and Protocols of PLCs for Computation Offloading at Field Level', *Cornwall University, CoRR* **abs/2106.04517**. https://arxiv.org/abs/2106.04517.

Hernandez, D. R. (2021), 'The Model View Controller Pattern – MVC Architecture and Frameworks Explained', https://www.freecodecamp.org/news/the-model-view-controller-pattern-mvc-architecture-and-frameworks-explained/. [Online; accessed 5 Mar-2024].

IEC61499 (2024), 'Standard for Distributed Automation", IEC 61499 Standard for Distributed Automation - Welcome.', https://iec61499.com/. [Online; accessed 26 Feb-2024].

Kadlecsik, T. (2021), 'Async Await in Node.js - How to Master it? –, Risingstack Engineering', https://blog.risingstack.com/mastering-async-await-in-nodejs/. [Online; accessed 4 Mar-2024].

Karlström, J. (2016), The WebSocket Protocol and Security : Best Practices and Worst Weaknesses, *in* 'CorpusID: 63836626 – Semantic Scholar, Computer Science'. https://api.semanticscholar.org/CorpusID:63836626.

Knapp, E. D. & Langill, J. T. (2015), 'Chapter 6 - Industrial Network Protocols', *Industrial Network Security, Syngress. (Second Edition)* pp. 121–169. https://doi.org/10.1016/B978-0-12-420114-9.00006-X.

Kolade, C. (2022), 'MVC in Computer Science – The MVC Model', https://www.freecodecamp.org/news/what-does-mvc-mean-in-computer-science/. [Online; accessed 6 Mar-2024].

Kuosmanen, H. (2016), Security Testing of WebSockets, *in* 'Corpus ID: 63056828 – Semantic Scholar, Computer Science'. https://api.semanticscholar.org/CorpusID:63056828.

Leung, C. W. (2013), 'Architecture of Distributed Real-Time Embedded System', *KTH Information and Communication Technology* . https://www.diva-portal.org/smash/get/diva2:688753/FULLTEXT02.

Li, J., Zhou, Y., Zhang, X., Liu, S. & Li, Q. (2021), 'Assessment of Industrial Internet Platform Application in Manufacturing Enterprises: System Construction and Industrial Practice', *IEEE Access* **9**, 103709–103727.

Liu, C. & Wang, K. (2012), An Online Examination System Based on UML Modeling and MVC Design Pattern, *in* '2012 International Conference on Control Engineering and Communication Technology', pp. 815–817.

Liu, Q. & Sun, X. (2012), Research of Web Real-Time Communication Based on Web Socket, *in* 'International Journal of Communications, Network and System Sciences', Vol. 5, pp. 797–801.

Mbed. (2024), 'Websockets -– Cookbook. mbed, os.mbed.com', https://os.mbed.com/cookbook/Websockets. [Online; accessed 7 Mar-2024].

MDN (2024), 'Express - Node Introduction - Learn Web Development', https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction. [Online; accessed 28 Aug-2024].

Murley, P., Ma, Z., Mason, J., Bailey, M. & Kharraz, A. (2021), WebSocket Adoption and the Landscape of the Real-Time Web, *in* 'Proceedings of the Web Conference 2021, pages =', Association for Computing Machinery.

Musings., R. (2020), 'Setting up NodeRed', https://blog.fuzzymistborn.com/setting-up-nodered/. [Online; accessed 26 Mar-2024].

Nagarajan, V. (2023), 'Understanding the Node.js Event Loop', https://medium.com/@vickypaiyaa/understanding-the-node-js-event-loop-23858526b2ff. [Online; accessed 3 Mar-2024].

NodeJS. (2024), 'Discover JavaScript Timers', https://nodejs.org/en/learn/asynchronous-work/discover-javascript-timers. [Online; accessed 29 Aug-2024].

Nodejs.org. (2024), 'An Introduction to the NPM Package Manager', https://nodejs.org/en/learn/getting-started/an-introduction-to-the-npm-package-manager. [Online; accessed 5 Mar-2024].

Npmjs. (2024*a*), 'npm – About, npm –inc', https://www.npmjs.com/about. [Online; accessed 5 Mar-2024].

Npmjs. (2024*b*), 'ws: a Node.js WebSocket library', https://www.npmjs.com/package/ws. [Online; accessed 01 Aug-2024].

Pang, C. & Vyatkin, V. (2007), Towards Formal Verification of IEC61499: Modelling of Data and Algorithms in NCES, *in* '2007 5th IEEE International Conference on Industrial Informatics', Vol. 2, pp. 879–884.

Pang, C., Yan, J. & Vyatkin, V. (2015), 'Time-Complemented Event-Eriven Architecture for Distributed Automation Systems', *IEEE Transactions on Systems, Man, and Cybernetics: Systems* **45**(8), 1165–1177.

Patrou, M., Kent, K. B., Siu, J. & Dawson, M. (2021), Energy and Runtime Performance Optimization of Node.js Web Requests, *in* '2021 IEEE International Conference on Cloud Engineering (IC2E)', pp. 71–82.

Peterson, D. (2022), 'The Origin Story of the PLC – Control Automation', https://control.com/technical-articles/the-origin-story-of-the-plc/. [Online; accessed 22 Feb-2024].

Pi4J. (2019), 'Pin Numbering - Raspberry Pi Zero W', https://www.pi4j.com/1.2/pins/model-zerow-rev1.html. [Online; accessed 02 Sept-2024].

PiHut. (2024), 'Raspberry Pi - ADC Pi', https://thepihut.com/products/adc-pizero. [Online; accessed 6 May-2024].

Qveflander, N. (2010), Pushing Real Time Data Using HTML Web Sockets, *in* 'Corpus ID:60958950 – Semantic Scholar, Computer Science'. https://api.semanticscholar.org/CorpusID:60958950.

Racchetti, L., Fantuzzi, C., Tacconi, L. & Bonfè, M. (2015), Towards an Abstraction Layer for PLC Programming Using Object-Oriented Features of IEC61131-3 Applied to Motion Control, *in* 'IECON 2015 - 41st Annual Conference of the IEEE Industrial Electronics Society', pp. 298–303.

Radix (2024), 'Node.js Statistics: Latest Usage Insights and Trends 2024', https://radixweb.com/blog/nodejs-usage-statistics. [Online; accessed 3 Mar-2024].

Raspberry-Pi. (2023), 'Raspberry Pi 5', https://datasheets.raspberrypi.com/rpi5/raspberry-pi-5-product-brief.pdf. [Online; accessed 2 May-2024].

Sehr, M. A., Lohstroh, M., Weber, M., Ugalde, I., Witte, M., Neidig, J., Hoeme, S., Niknami, M. & Lee, E. A. (2021), 'Programmable Logic Controllers in the Context of Industry 4.0', *IEEE Transactions on Industrial Informatics* **17**(5), 3523–3533.

Selişteanu, D., Roman, M., Şendrescu, D., Petre, E. & Popa, B. (2018), A distributed control system for processes in food industry: Architecture and implementation, *in* '2018 19th International Carpathian Control Conference (ICCC)', pp. 128–133.

Shabani, I. (2018), 'Implementation of Service Oriented Architectures in Civil Engineering', *International Journal of Civil Engineering and Technology* **9**, 707–719.

Shah, H. & Soomro, T. (2017), 'Node.js Challenges in Implementation', *Global Journal of Computer Science and Technology* **17**, 72–83.

Skvorc, D., Horvat, M. & Srbljic, S. (2014), Performance Evaluation of Websocket Protocol for Implementation of Full-Duplex Web Streams, *in* '2014 37th International

Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)', pp. 1003–1008.

Statista (2022), 'Most Used Web Frameworks Among Developers Globally 2020', https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/. [Online; accessed 3 Mar-2024].

Sverdlov, E. (2024), 'How To Create a New User and Grant Permissions in MySQL', https://www.digitalocean.com/community/tutorials/how-to-create-a-new-user-and-grant-permissions-in-mysql. [Online; accessed 30 Aug-2024].

Tomasetti, M. (2021), 'An Analysis of the Performance of Websockets in Various Programming Languages and Libraries', *Computer Science – Rampo College of New Jersey* .

TP-Link (2024), '5-Port Desktop Switch', https://www.tp-link.com/au/business-networking/unmanaged-switch/tl-sg105/. [Online; accessed 25 July-2024].

Umer, M., Mahesh, B., Hanson, L., Khabbazi, M. & Onori, M. (2018), 'Smart Power Tools: An Industrial Event-Driven Architecture Implementation', *Procedia CIRP* **72**, 1357–1361. 51st CIRP Conference on Manufacturing Systems.

Vargas, H., Sánchez, J., Jara, C., Candelas, F., Torres, F. & Dormido, S. (2011), 'A Network of Automatic Control Web-Based Laboratories', *IEEE Transactions on Learning Technologies* **4**(3), 197–208.

Wang, H., Luo, H., Jiang, Y., Xu, X. & Li, X. (2022), A data-driven distributed control method for performance optimization of interconnected industrial processes, *in* '2022 4th International Conference on Industrial Artificial Intelligence (IAI)', pp. 1–6.

Appendix A

Project Specification

The project specification is provided on the following pages.

ENP4111 Specification and Work Plan

Title: Investigation of Web-Based Technologies in the Development of Full-Stack, Small to Mid-Sized Distributed Industrial Control Systems

Name : Scott Carr

Student ID : [redacted]

Supervisor: John Leis

Introduction and Background

The focus of this research project is to perform an in-depth review of existing, modern web-based software and technology and its direct application in the development of full-stack solutions for small to mid-sized distributed control systems (DCS) in the industrial sector.

The intent is to highlight benefits of such full end-to-end implementations including basing hardware on embedded microprocessors as opposed to standard industry programmable logic controllers (PLC), creating communication networks based on the Websocket protocol with the intent to meet real-time constraints and linking data to bespoke web-page based Human-Machine Interface (HMI) methods.

Combined, these factors could offer heavy cost reductions in design and implementation, flexible solutions and better access to captured data. Along with such potential benefits however the purpose of this research process will be assessing how such implementations can be achieved and what performance-based thresholds must be met to use such systems reliably and efficiently.

Distributed Control Systems underpin most manufacturing and process plant operations, being the backbone which control the equipment and harness the sensor data to ensure safe, reliable and efficient operation across industries worldwide. These systems are made up of multiple system components which all must meet design specifications and performance guidelines to maintain operations for the companies and entities that utilise them.

As the transition to Industry 4.0 develops, a major shift is occurring in the way that data is managed, systems are implemented in line with trends to make data more accessible and utilise sensor information with more flexible control implementations. In the last decade or so web-based interfaces have become more frequently used for interacting with hardware controllers and smart devices. Novel and streamlined solutions have been explored for these instances such as in the research paper *'Web-Based Human-Machine Interfaces of Industrial Controllers in Single-Page Applications'* by Shyr-Long Jeng et al whereby the display of key data is achieved using web technologies linked to a traditional programmable logic controller (PLC).

Further evidence of such work exists in the paper *'Efficient Web-based Monitoring and Control System'*, where Ahmed M. Mohamed and Hosny A. Abbas show a methodology of implementing web-based technologies for a supervisory control and data acquisition (SCADA) scheme again connected to a PLC.

While these examples show some benefits over legacy approaches there is limited research or evidence available of these technologies extending into the industrial controller realm to allow a front to back-end, full-stack infrastructure to be explored.

One area where full-stack, embedded solutions of similar technologies have seen rapid growth and experimental development is in the Internet of Things (IOT) and home automation space. Work in this area has shown such approaches can yield highly adaptable, cost-effective, and reliable systems especially for single sensor systems and the like as explored in *'Smart Home Security Application Enabled by IoT'* by C. Davidson et al.

Edge computing has also seen web-based technologies used which can assist in bridging the layer between hardware-based control layers and higher-level communication layers for data transfer, this is explained in the article *'Towards Smart Home Automation Using IoT-Enabled Edge-Computing Paradigm'* by Hikmat Yar et al.

Further to these lower-level control solutions, more complex interactive solutions for this sector have been developed to increase connectivity using web-based communication systems such as those detailed in *'Study on Integration Technologies of Building Automation Systems based on Web Services'* by Jianbo Bai et al, where co-ordinated distributed networks are created and linked together. Cumulatively, these papers show that there exists a breadth of applicable reference material where web-based solutions are implemented for solutions aligning with or closely relating to distributed, industrial based applications.

Objectives and Aims

There are several key objectives when looking to implement a full-stack industrial control solution using web-based technologies such as whether they have the performance reliability, real-time capability, and deterministic behaviour traits to suit such implementations.

This research project aims to review core aspects of modern distributed control systems with a focus on small to mid-sized systems and carry out research as to whether web-based technologies may provide adequate solutions to build the core elements of such systems.

These elements include control algorithms & modules, interfacing software, database structures, front-end displays, comms and middleware layers, monitoring and alarm management and other aspects. These further aspects include features such as authentication and remote connectivity and other higher-level functionality which may not be actively explored within the scope of this project.

The core software will rely on a mix of JavaScript programming language implemented on the runtime NodeJS, MySql databasing, the Websockets Protocol and other features.

Specific Objectives:

- Conduct a detailed review of the key components in mid-sized control systems and detail how the selected software can be utilised to develop both theoretical, and where possible, practical solutions of each equivalent section.
- Implement and test performance of a Websocket based communication layer over multiple nodes to allow distributed data channels and control to be realised.
- Review and assess how an implementation of the NodeJS server runtime can be structured to meet potential schedule deadlines and deliver reliable core control system operation.
- Develop sound, re-useable modular code and class-based systems for any control system architectures created.

Expected Outcomes:

- Determine the suitability and benefits of a full-stack JavaScript based control system.
- Report and display the performance-based results for Websocket based communication developed to meet the distributed architecture.
- Clearly document the methods used to create and implement control system components and where the key benefits are gained in the process.
- Show that low-cost hardware with open-source software can build up the layers of a performant control system.
- Detail how data can become more accessible and tailored with a full-stack approach.
- Provide recommendations on further improvements and future developments using such technologies.

Work Plan & Timeline

See following page for project plan & timeline.

[redacted]

Project Name	Final Year ENG4111 Project	Student Name	Scott Carr	Student Number	U0061045137	Start Date	5/02/2024	End Date	6/12/2024	Overall Progress	0%	Project Schedule

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
Tasks	Status	29-Jan	5-Feb	12-Feb	19-Feb	26-Feb	4-Mar	11-Mar	18-Mar	25-Mar	1-Apr	8-Apr	15-Apr	22-Apr	29-Apr	6-May	13-May	20-May	27-May	3-Jun	10-Jun	17-Jun	24-Jun	1-Jul	8-Jul	15-Jul	22-Jul	29-Jul	5-Aug	12-Aug	19-Aug	26-Aug	2-Sep	9-Sep	16-Sep	23-Sep	30-Sep	7-Oct	14-Oct	21-Oct	28-Oct	4-Nov	11-Nov	18-Nov	25-Nov	2-Dec	9-Dec	16-Dec
Approval / Start of Project																																																
Resourcing Prep	Started	■	■	1																																												
Drafting of Dissertation	Not started				■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	2W Holiday		■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■		
Literature Review & Research Design	Not started				■	■	■	■	■	■																																						
Hardware Config.																																																
Review hardware concepts	Not started										2																																					
Develop Hardware methodology	Not started											■	■																																			
Software Config.																																																
Review software concepts	Not started													■	■																																	
Develop software methodology	Not started															■	■																															
System Development & Testing																																																
Setup OS / Servers / software etc	Not started																	3	■	■																												
Develop software stack components	Not started																				■	■	■	■	2W Holiday																							
Setup Websockets comms layers	Not started																										■	■																				
Testing of control, comms & data layers	Not started																												■	■	■																	
Webpage Elements & HMI	Not started																															■	■															
Results collation & Drafting																																																
Draft Dissertation Submission	Not started																																	4														
Continue testing & review as reqd	Not started																																		■	■	■											
Collate results	Not started																																					■										
Project Finalisation																																																
Project Presentation	Not started																																						5	6								
Final Dissertation Paper																																									■	7	■	■	■	■		
Delivery of Project																																																
Specification & Work Plan Submission				1																																												
					Literature Review Submission						2																																					
												Methodology Submission						3																														
																			Draft Dissertation and Preliminary Results Submission															4														
																																			Presentation				5									
																																				Reflection Submission				6								
																																						Dissertation Submission				7						

Resources Required:

- **Equipment:**
  - Raspberry Pi 5/ Pi Zero W / ADC Pi embedded systems.
  - Home tools, connectors, crimpers, comms cables as reqd.
  - Laptop, monitors, general computing equipment.
- **Software:**
  - Open-source software: NodeJS/ mySql / JS / Socket.io etc
  - Open-source data testing software.
  - Use MS Excel / JS or other software for data analysis & reporting.
  - EndNote for citations, Ms word for dissertation etc.
- **Access:**
  - ScienceDirect, IEEE Explore and other UniSQ associated access for research and literature review.
  - Textbooks and similar resources as required for any practical based information and processes.

References:

Jeng, S.-L., Chieng, W.-H., Chen, Y., 2021. *'Web-Based Human-Machine Interfaces of Industrial Controllers in Single-Page Applications'*. Mobile Information Systems 2021, 1–13. . Available at: https://doi.org/10.1155/2021/6668843 [Citations: 9].

Mohamed, A.M. and Abbas, H.A., 2011, May. *'Efficient web-based monitoring and control system'*. In Proceedings of the Seventh International Conference on Autonomic and Autonomous Systems, ICAS (pp. 22-27). [Citations: 14]

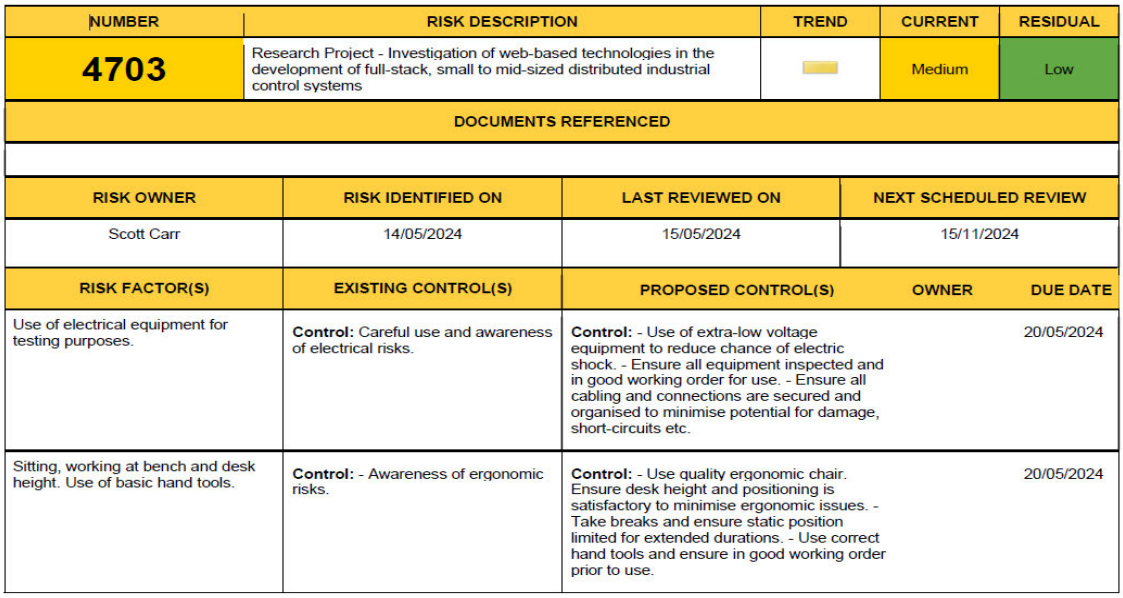
Davidson, C., Rezwana, T. and Hoque, M.A., 2019. *'Smart home security application enabled by IOT: Using arduino, raspberry pi, nodejs, and mongodb'*. In Smart Grid and Internet of Things: Second EAI International Conference, SGIoT 2018, Niagara Falls, ON, Canada, July 11, 2018, Proceedings 2 (pp. 46-56). [Citations: 19]

Yar, H., Imran, A.S., Khan, Z.A., Sajjad, M., Kastrati, Z., 2021. *'Towards Smart Home Automation Using IoT-Enabled Edge-Computing Paradigm'*. Sensors 21, 4932. Available at: https://doi.org/10.3390/s21144932 [Citations: 101]

Bai, J., Xiao, H., Yang, X., Zhang, G., 2009. *'Study on integration technologies of building automation systems based on web services'*. 2009 ISECS International Colloquium on Computing. Available at: https://doi.org/10.1109/cccm.2009.5267730 [Citations: 29]

Appendix B

Risk Assessment



NUMBER	RISK DESCRIPTION	TREND	CURRENT	RESIDUAL
4703	Research Project - Investigation of web-based technologies in the development of full-stack, small to mid-sized distributed industrial control systems	🟨	Medium	Low

DOCUMENTS REFERENCED

RISK OWNER	RISK IDENTIFIED ON	LAST REVIEWED ON	NEXT SCHEDULED REVIEW
Scott Carr	14/05/2024	15/05/2024	15/11/2024

RISK FACTOR(S)	EXISTING CONTROL(S)	PROPOSED CONTROL(S)	OWNER	DUE DATE
Use of electrical equipment for testing purposes.	**Control:** Careful use and awareness of electrical risks.	**Control:** - Use of extra-low voltage equipment to reduce chance of electric shock. - Ensure all equipment inspected and in good working order for use. - Ensure all cabling and connections are secured and organised to minimise potential for damage, short-circuits etc.		20/05/2024
Sitting, working at bench and desk height. Use of basic hand tools.	**Control:** - Awareness of ergonomic risks.	**Control:** - Use quality ergonomic chair. Ensure desk height and positioning is satisfactory to minimise ergonomic issues. - Take breaks and ensure static position limited for extended durations. - Use correct hand tools and ensure in good working order prior to use.		20/05/2024

Figure B.1: UniSQ Risk Assessment for the project.

Appendix C

Ethical Clearance

There are no Ethical Clearances applicable to this project.

Appendix D

Supporting Information

D.1 Introduction to this Appendix

This Appendix contains all supporting information associated with the project development and includes the following sections:

- Further Technical Information and Specifications
- Installations and Configurations
- Software Code Files

D.2 Further Technical Information and Specifications

The following Technical Specifications are given for the Raspberry Pi Five and Raspberry Pi Zero W respectively.

A list of the specifications are given below for the Raspberry Pi Five:

- VideoCore VII GPU, supporting OpenGL ES 3.1, Vulkan 1.2
- Dual 4Kp60 HDMI® display output with HDR support
- 4Kp60 HEVC decoder
- LPDDR4X-4267 SDRAM
- Dual-band 802.11ac Wi-Fi®
- Bluetooth 5.0 Low Energy (BLE)
- microSD card slot, with support for high-speed SDR104 mode
- 2 × USB 3.0 ports, supporting simultaneous 5Gbps operation
- 2 × USB 2.0 ports
- Gigabit Ethernet, with PoE+ support
- 2 × 4-lane MIPI camera/display transceivers
- 5V/5A DC power via USB-C, with Power Delivery support
- Raspberry Pi standard 40-pin header
- Real-time clock (RTC), powered from external battery

(Raspberry-Pi. 2023)

A list of the specifications are given below for the Raspberry Pi Zero W:

- BCM 2835 SoC (ARM11 at 1GHz)
- 512MB of RAM
- On-board Wi-Fi - 2.4 GHz 802.11 b/g/n (BCM43438)
- On-board Bluetooth 4.1 + HS Low-energy (BLE) (BCM43438)
- Storage: micro-SD
- Display: mini-HDMI
- Power: USB micro-B
- USB: 1 x USB micro-B
- CSI camera connector (requires adaptor cable)
- Unpopulated 40-pin GPIO connector (requires soldering)
- Compatible with existing pHAT/HAT add-ons
- Dimensions: 65mm x 30.5mm x 5mm
- Power Supply Voltage Requirement: 5.1V

(Core-Electronics. 2024)

D.3 Installations and Configurations

The following subsections detail the steps involved to setup the embedded platform operating systems and base applications including dependencies. From this point the software was able to be developed to meet the project objectives as detailed in the Design of the Project Section 4.

D.3.1 Operating systems, Installations and Runtime

To allow development of each NodeJS application a number of initial installation and setup steps were required. Each of the three MCU's needed an operating system to be downloaded and then flashed as a software image onto their respective Micro-SD card. These were then installed into the Micro-SD card holder on the Raspberry Pi MCU.

The operating systems chosen for the server and client MCU's are detailed below:

- Pi Five Server: Ubuntu 23.10, 'ubuntu-23.10-preinstalled-desktop-arm64+raspi.img'
- Pi Zero Clients: Raspian Bookworm, '2023-12-05-raspios-bookworm-armhf.img'

These OS versions were downloaded from 'www.ubuntu.com/downloads/raspberry-pi', and 'www.raspberrypi.com/software/operating-systems/#raspberry-pi-os-32-bit', respectively. These programs were loaded using the BalenaEtcher imaging tool, 'balenaEtcher-Setup-1.18.11', which was downloaded from 'https://etcher.balena.io'.

Once loaded, the MCU's were powered and basic system configuration steps were taken to enable functionality. This included the following steps:

1. Perform initial Raspberry Pi and Ubuntu account registration of each unit. To do this there were prompts built into the system. System prompts required the below information & settings:
   - Setting of keyboard layout to English (US)
   - Setup of wireless connectivity to home router to allow remote downloads
   - Setting of country to Australia with timezone to Perth

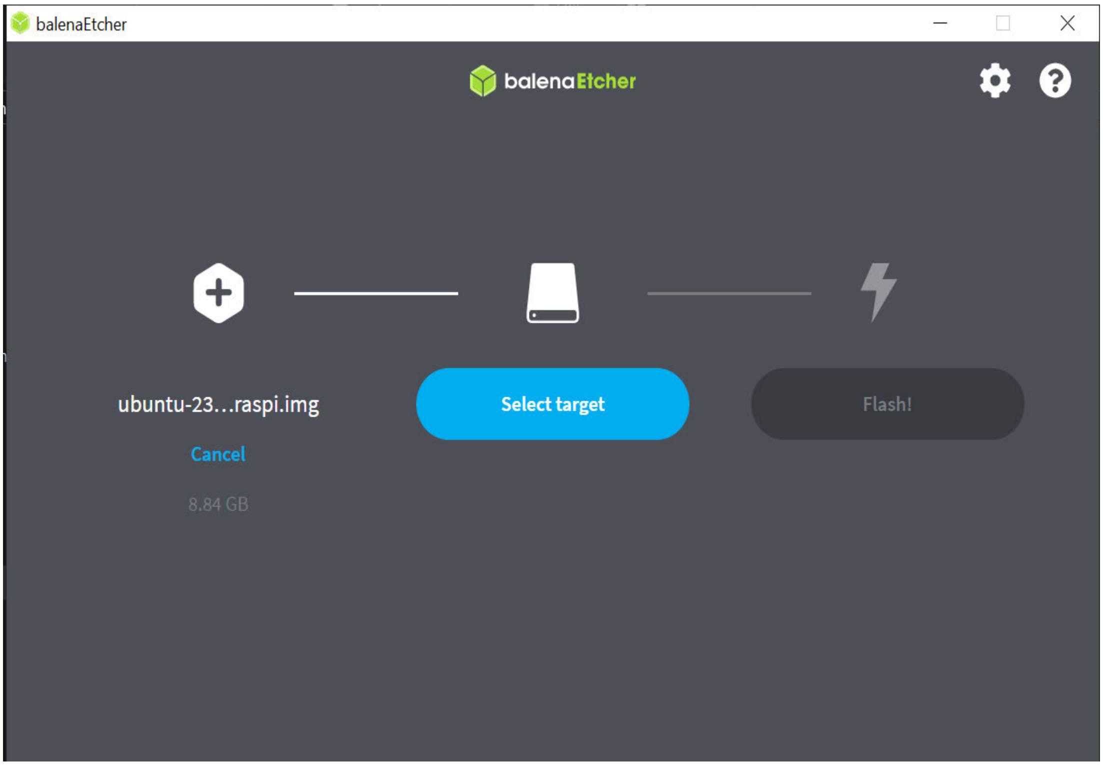


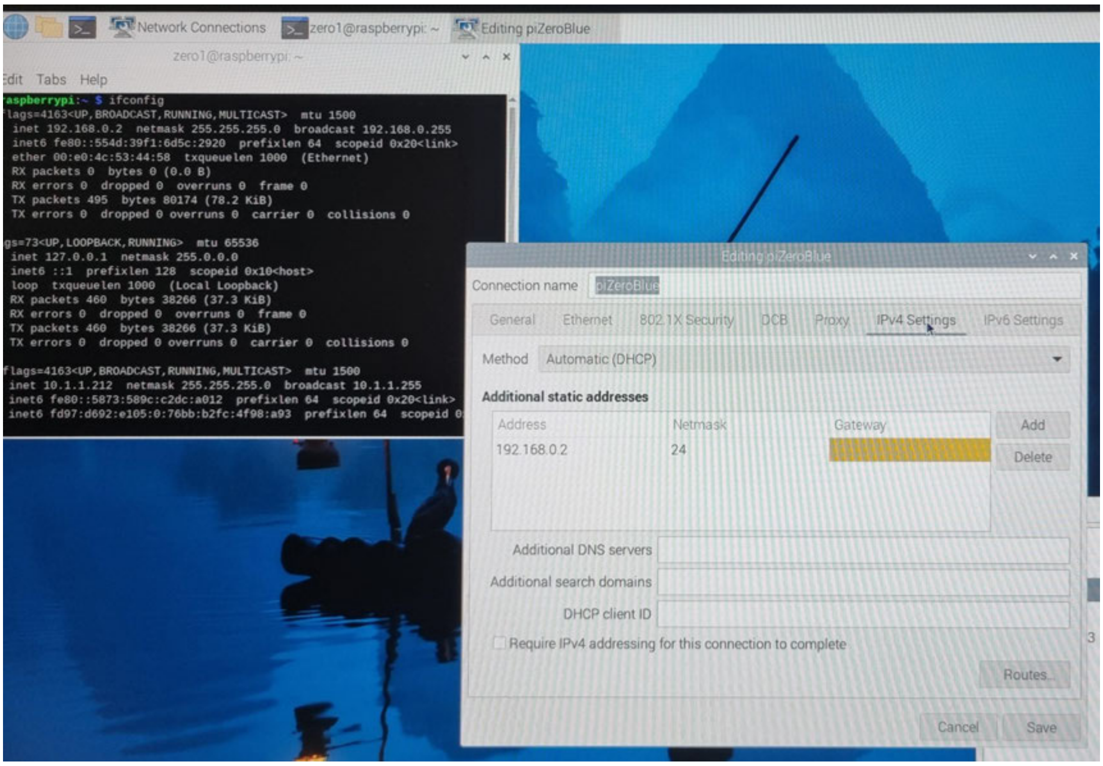
Figure D.1: Image of the Etcher process to flash the Micro-SD's with OS image.

- Creating a unique username with password for each of the 3 units
- Update of Software to bring the flashed OS up to currency

2. Creation of wired connection on each unit for Local Area Network Ethernet connections. This required navigation to the Network Connection tab, then opening the wired connections tab and creating a new IPv4 static address to ensure each network address would not change after system shutdown.

3. Download of the net-tools package via terminal using: 'sudo apt install net-tools'. This allowed use of valuable network diagnostic tools for use within terminal such as ifconfig, arp -a and netstat -a. These were used to confirm the static IP's assigned were working to assist with low-level network setup as detailed in Section 5.2.2.

4. Enabling of the Raspberry Pi lower level configuration settings via the preferences menu. This was completed to allow use of the GPIO pins as input / outputs and to allow the I2C communication bus to be used for reading the ADC Pi voltages.



```
aspberrypi:~ $ ifconfig
lags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
 inet 192.168.0.2  netmask 255.255.255.0  broadcast 192.168.0.255
 inet6 fe80::554d:39f1:6d5c:2920  prefixlen 64  scopeid 0x20<link>
 ether 00:e0:4c:53:44:58  txqueuelen 1000  (Ethernet)
 RX packets 0  bytes 0 (0.0 B)
 RX errors 0  dropped 0  overruns 0  frame 0
 TX packets 495  bytes 80174 (78.2 KiB)
 TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

gs=73<UP,LOOPBACK,RUNNING>  mtu 65536
 inet 127.0.0.1  netmask 255.0.0.0
 inet6 ::1  prefixlen 128  scopeid 0x10<host>
 loop  txqueuelen 1000  (Local Loopback)
 RX packets 460  bytes 38266 (37.3 KiB)
 RX errors 0  dropped 0  overruns 0  frame 0
 TX packets 460  bytes 38266 (37.3 KiB)
 TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
 inet 10.1.1.212  netmask 255.255.255.0  broadcast 10.1.1.255
 inet6 fe80::5873:589c:c2dc:a012  prefixlen 64  scopeid 0x20<link>
 inet6 fd97:d692:e105:0:76bb:b2fc:4f98:a93  prefixlen 64  scopeid 0
```

Editing piZeroBlue

Connection name: piZeroBlue

General | Ethernet | 802.1X Security | DCB | Proxy | IPv4 Settings | IPv6 Settings

Method: Automatic (DHCP)

Additional static addresses

Address	Netmask	Gateway
192.168.0.2	24	

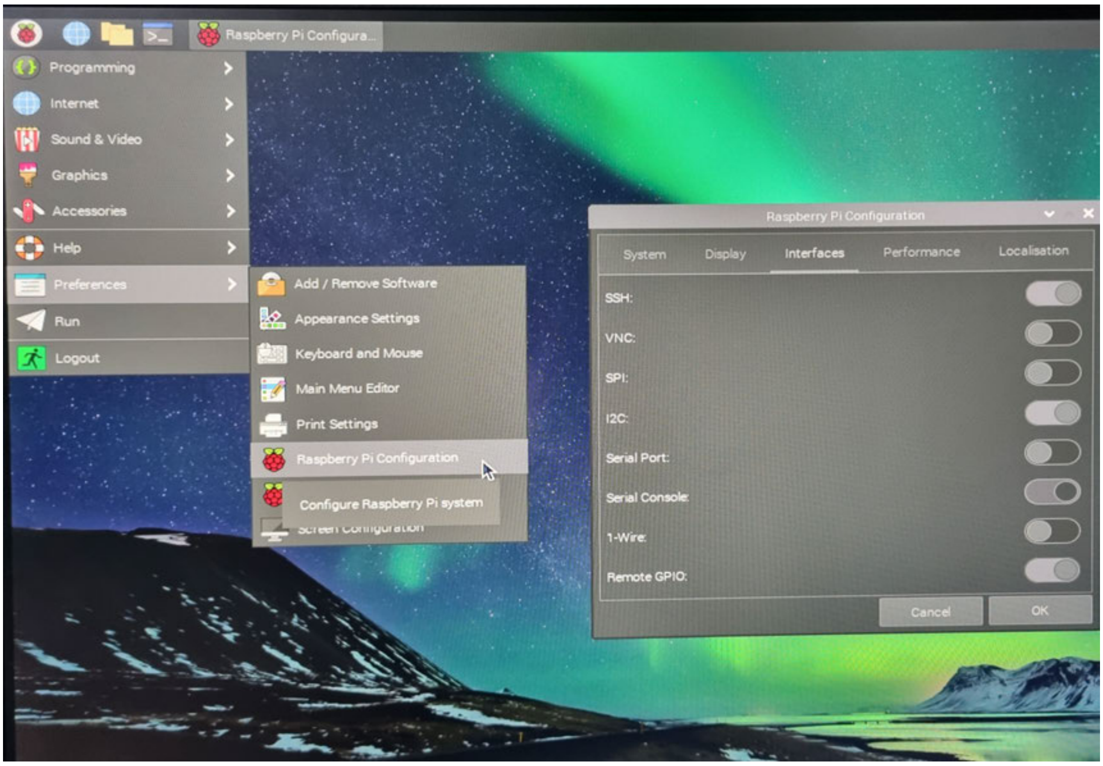
Additional DNS servers:

Additional search domains:

DHCP client ID:

□ Require IPv4 addressing for this connection to complete

Figure D.2: IPv4 configuration with ifconfig confirmation of settings.



Raspberry Pi Configuration

System | Display | Interfaces | Performance | Localisation

SSH:

VNC:

SPI:

I2C:

Serial Port:

Serial Console:

1-Wire:

Remote GPIO:

Figure D.3: Enabling configuration settings via preferences tab.

Once these steps were completed the associated systems were ready for NodeJS programming to be initialised and developed.

D.3.2 NodeJS Installation and Application Initialisation

Both the server and client units used a similar file architecture for the development of their respective NodeJS applications. During testing it was found that the Pi Zero W units do not support the chosen Visual Studio Code software editing suite. Therefore, the client programs were developed from a laptop session and with use of the Pi Zero W's onboard Geany lightweight program text editor. This still allowed efficient development with core program sections being transferred via USB flash drive to the units.

There were a number of packages necessary to allow development of the NodeJS applications. These were downloaded from a terminal session on each individual unit and are detailed below.

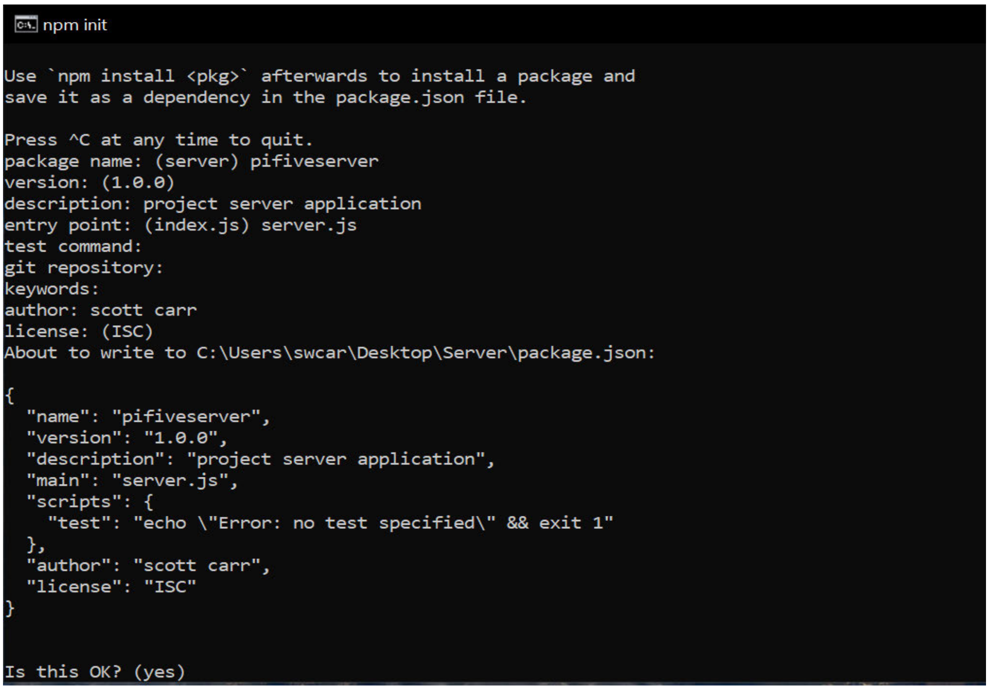
- Prior to install of packages a system update is run. 'sudo apt-get update'.
- Installation of NodeJS package. 'sudo apt install nodejs'.
- Installation of npm manager. 'sudo apt install npm'.
- Installation of Visual Studio Code (Pi Five only). 'sudo apt install code'.

When downloading packages, repeat issues were encountered due to incomplete file transfer. To remedy this, the command 'sudo dpkg-reconfigure -plow unattended-upgrades', was used which clears and completes partial file installations and updates. This is a common occurrence with terminal updates and the above command is a linux based solution.

Once npm was installed the base application on each device could be initialised. This was performed by creating a base folder which could be on the desktop, then opening a terminal session and navigating to within the folder. Once the folder path was set, entering 'npm init' in the terminal created an application. By following the prompts a base application was created ready for development as shown below.

Then further dependencies were added as below:

- The Express package was installed. ‘npm install –save express’.
- EJS middleware package was installed (Pi Five only). ‘npm install –save ejs’.
- Websockets package was installed. ‘npm install –save ws’.
- Body-Parser package was installed (Pi Five only). ‘npm install –save body-parser’.
- Node system performance package was installed. ‘npm install perf-hooks’.
- MySQL package was installed (Pi Five only). ‘npm install mysql’.

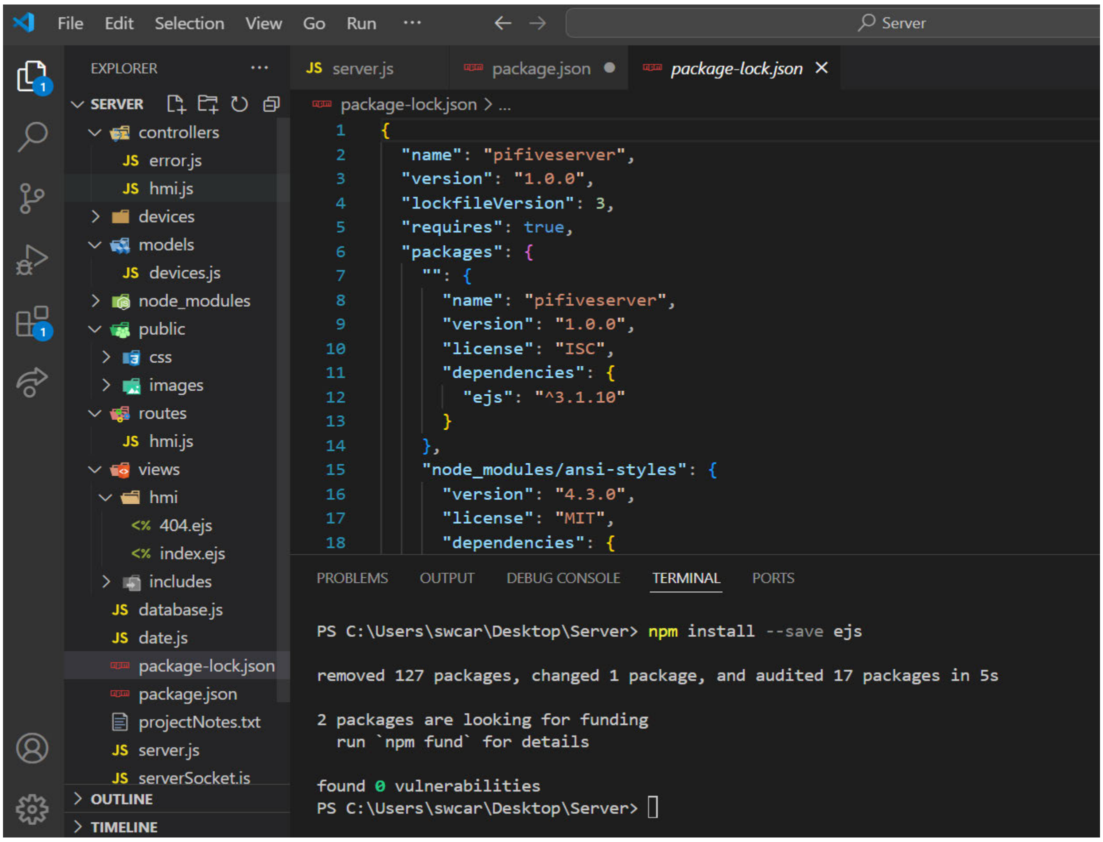


```
Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (server) pifiveserver
version: (1.0.0)
description: project server application
entry point: (index.js) server.js
test command:
git repository:
keywords:
author: scott carr
license: (ISC)
About to write to C:\Users\swcar\Desktop\Server\package.json:

{
  "name": "pifiveserver",
  "version": "1.0.0",
  "description": "project server application",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "scott carr",
  "license": "ISC"
}


Is this OK? (yes)
```

Figure D.4: Intialisation of NodeJS application using npm package manager command.

With the initialisation step and dependencies added, each device then had a backbone application structure with all necessary packages available. By using –save, these package dependencies were loaded into the npm package.json file which enabled the libraries to be called within the main node application script. Further folders and files could be added within the parent directory in which the application was created to build the system architecture for each device.

The final developed file and folder structure for the Server application is shown below. This shows how the application was structured within the ‘Server’ folder on the Pi Five desktop when running within VS Code. This also displays an example of how the EJS module was downloaded in the lower terminal section and subsequently placed into the package dependency list in the package-lock.json file.



```
{
  "name": "pifiveserver",
  "version": "1.0.0",
  "lockfileVersion": 3,
  "requires": true,
  "packages": {
    "": {
      "name": "pifiveserver",
      "version": "1.0.0",
      "license": "ISC",
      "dependencies": {
        "ejs": "^3.1.10"
      }
    },
    "node_modules/ansi-styles": {
      "version": "4.3.0",
      "license": "MIT",
      "dependencies": {
```

```
PS C:\Users\swcar\Desktop\Server> npm install --save ejs

removed 127 packages, changed 1 package, and audited 17 packages in 5s

2 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS C:\Users\swcar\Desktop\Server> 
```

Figure D.5: Visual Studio Code Server application and file / folder structure.

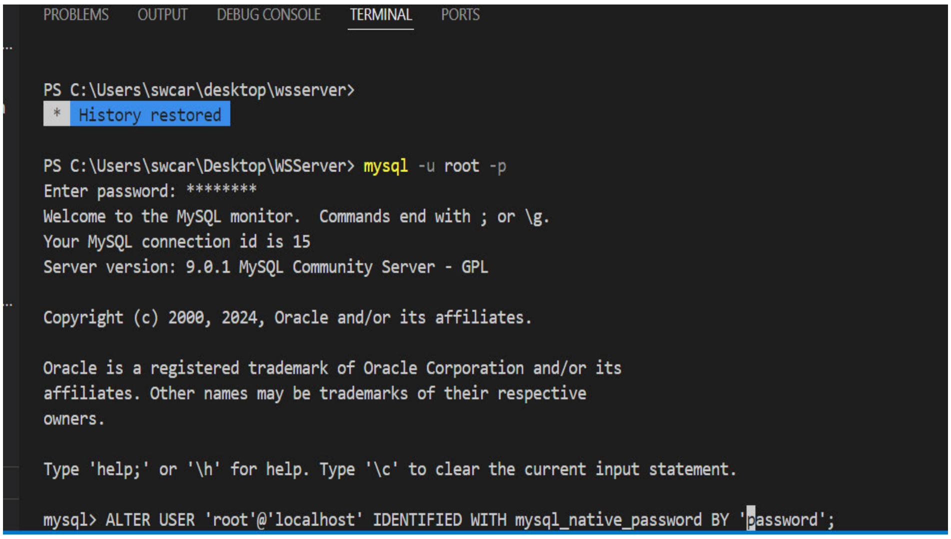
D.3.3 MySQL Installation and Database Initialisation

To install MySQL to the Pi Five, the terminal commands listed below, were run in order:

- ‘sudo apt update’
- ‘sudo apt upgrade’
- ‘sudo apt install mysql-server’

Once downloaded, the command ‘sudo mysql_secure_installation’ allows installation. Once installed, a unique user account name and password were created for connecting to the MySQL program. There were issues encountered which required setting of the MySQL system path, and privilege modifications to allow NodeJS to connect during the development. With such issues resolved the user and access configuration could be completed. To do this the following steps were taken:

1. Connect to mysql via terminal. First navigate to the directory, then type: ‘mysql -u root -p’. This prompted for password entry, once entered, a welcome message was displayed and the ‘mysql>’ prompt was shown, allowing entry of queries and commands. This is shown in the below Figure 4.17.



```
PS C:\Users\swcar\desktop\wsserver>
 *  History restored

PS C:\Users\swcar\Desktop\WSServer> mysql -u root -p
Enter password: ********
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 15
Server version: 9.0.1 MySQL Community Server - GPL

Copyright (c) 2000, 2024, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY 'password';
```

Figure D.6: Terminal login to MySQL program.

2. At the ‘mysql>’ prompt entered the below entries one by one:
   mysql>CREATE USER ‘scott’ ⓐ ‘localhost’ IDENTIFIED BY ‘scottpw’
   mysql>GRANT ALL PRIVILEGES ON * . * TO ‘scott’ ⓐ ‘localhost’ WITH GRANT OPTION
   mysql>FLUSH PRIVILEGES
   (Sverdlov 2024)

3. Once this was completed the MySQL program was ready to connect via the server NodeJS application. This was achieved through the interface software developed in the Functional Design Section.

The database account details were:

- Name: piFiveDB
- Host: localhost
- User: scott
- Password: [redacted]
- Port: 3306

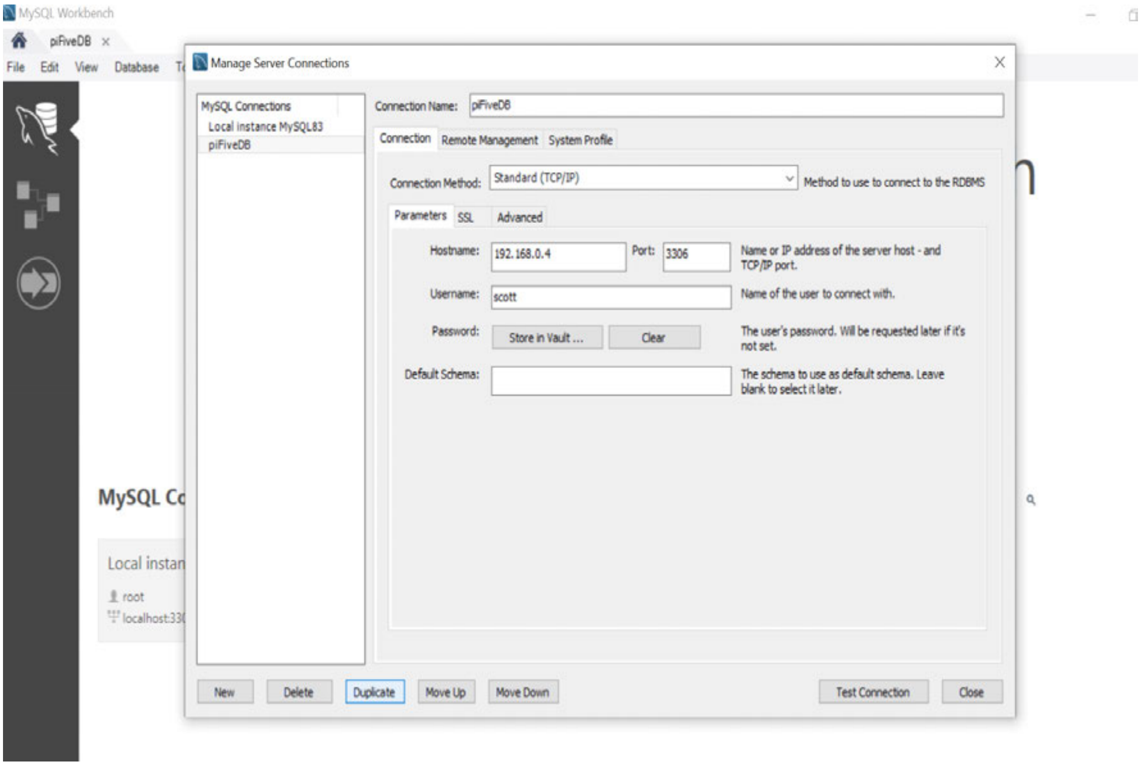
The tables created for this project were:

- alarmdata - Logged the tag / status / clienttime parameters
- client1iodata - Logged client 1 analog and discrete device tag / value / servertime / clienttime parameters.
- client1sysdata - Logged client 1 system data with tag / value / clienttime.
- client2iodata - Logged client 2 analog and discrete device tag / value / servertime / clienttime parameters.
- client2sysdata - Logged client 2 system data with tag / value / clienttime.
- serversysdata - Logged server system data with tag / value / servertime.

MySQL WorkBench 8.0

To allow connection, monitoring, export and review of database data, MySQL Workbench was installed onto the laptop as part of the project. This was downloaded from: mysql-workbench-community-8.0.36-winx64 and allowed a visual platform to interact with the stored Pi Five database.

Once the program was downloaded the database connection was made via a USB to ethernet connector from Laptop to the 5-Port switch which the Pi Five was also connected to. The laptop was set to the static IP 192.168.0.5, then the connection was established as shown below.



Connection Name: piFiveDB

Connection Method: Standard (TCP/IP) Method to use to connect to the RDBMS

Hostname: 192.168.0.4 Port: 3306 Name or IP address of the server host - and TCP/IP port.

Username: scott Name of the user to connect with.

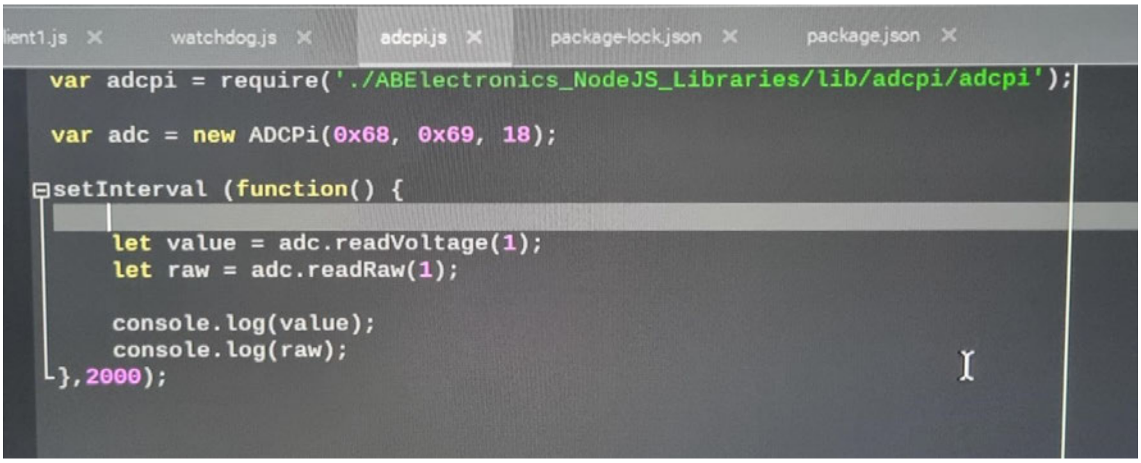
Password: The user's password. Will be requested later if it's not set.

Default Schema: The schema to use as default schema. Leave blank to select it later.

Figure D.7: Connection to MySQL database via Workbench 8.0.

D.3.4 ADC Pi interface card, setup and testing

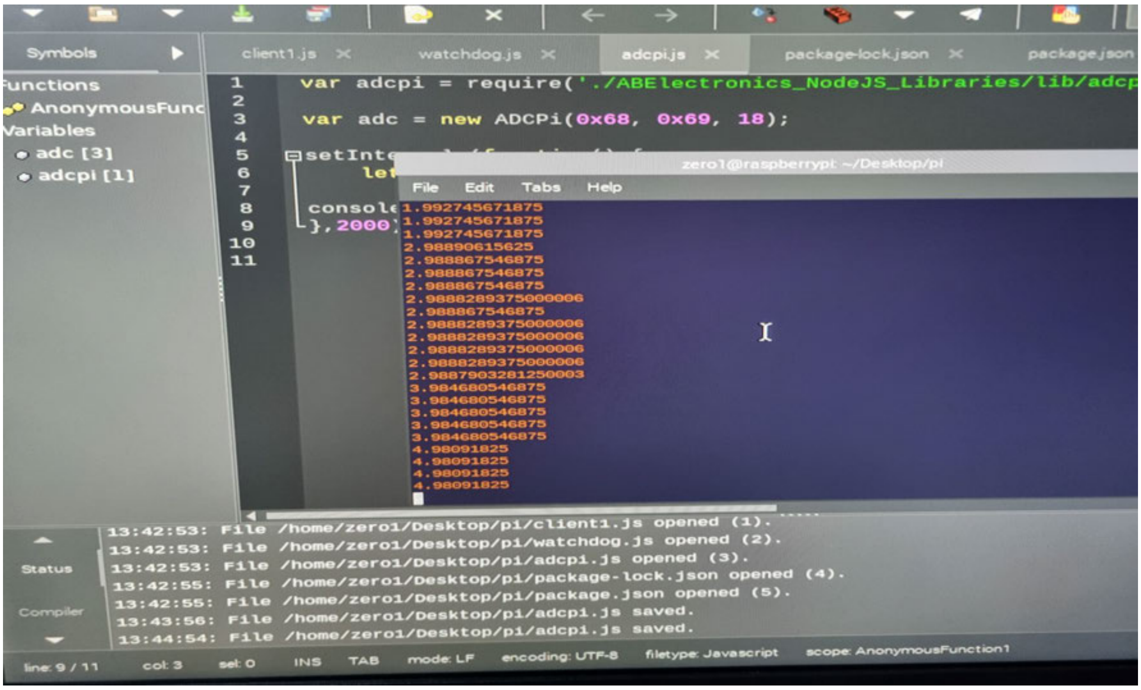
The ADC Pi board was enabled by importing the ‘AB Electronics ’ library which has in-built NodeJS support. Then the units library class can be assigned as an object with the two ADC on-board chips assigned unique I2C hexadecimal addresses listed and the resolution.



```
var adcpi = require('./ABElectronics_NodeJS_Libraries/lib/adcpi/adcpi');

var adc = new ADCPi(0x68, 0x69, 18);

setInterval (function() {

    let value = adc.readVoltage(1);
    let raw = adc.readRaw(1);

    console.log(value);
    console.log(raw);
},2000);
```

Figure D.8: Code used to test ADC Pi interface with NodeJS

Once the object is created it can be called using the adc.ReadVoltage(x) or adc.readRaw(x) with ‘x ’ representing the channel from 1 to 8. Then the scanned voltage or raw value can be assigned to a variable to be used within the program such as client1iodata().



```
var adcpi = require('./ABElectronics_NodeJS_Libraries/lib/adcp

var adc = new ADCPi(0x68, 0x69, 18);

setInte
    let

console
},2000
```

```
1.992745671875
1.992745671875
1.992745671875
2.98890615625
2.988867546875
2.988867546875
2.988867546875
2.9888289375000006
2.988867546875
2.9888289375000006
2.9888289375000006
2.9888289375000006
2.9888289375000006
2.9887903281250003
3.984680546875
3.984680546875
3.984680546875
3.984680546875
3.984680546875
4.98091825
4.98091825
4.98091825
4.98091825
```

```
13:42:53: File /home/zero1/Desktop/pi/client1.js opened (1).
13:42:53: File /home/zero1/Desktop/pi/watchdog.js opened (2).
13:42:53: File /home/zero1/Desktop/pi/adcpi.js opened (3).
13:42:55: File /home/zero1/Desktop/pi/package-lock.json opened (4).
13:42:55: File /home/zero1/Desktop/pi/package.json opened (5).
13:43:56: File /home/zero1/Desktop/pi/adcpi.js saved.
13:44:54: File /home/zero1/Desktop/pi/adcpi.js saved.
```

```
line: 9 / 11    col: 3    sel: 0    INS    TAB    mode: LF    encoding: UTF-8    filetype: Javascript    scope: AnonymousFunction1
```

Figure D.9: Program scaled output whilst scanning input.

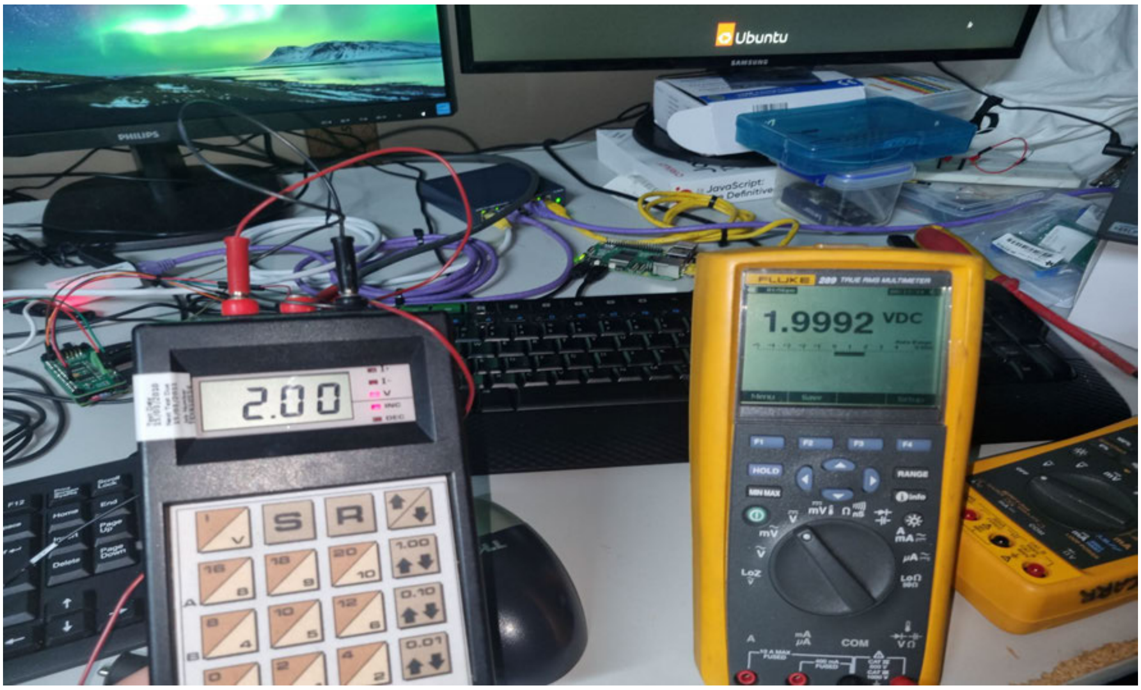
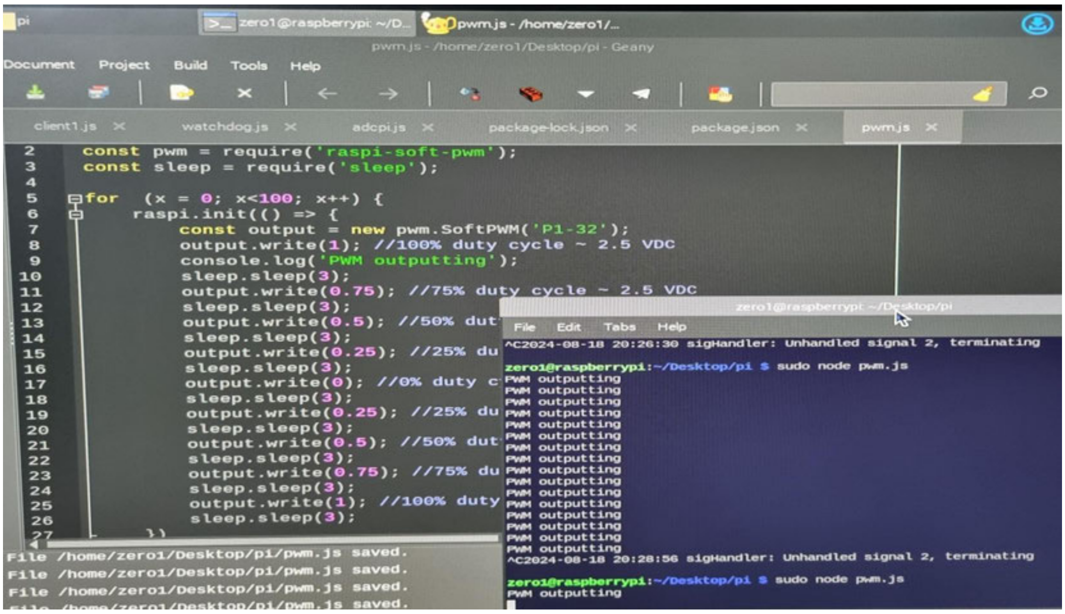


Figure D.10: Test equipment with readings during tests

D.3.5 Pulse Width Modulation Output, setup and testing

The code for the Pulse-Width Modulation output testing was developed from the NodeJS ‘raspi-soft-pwm’ library. This allowed easy to implement PWM operations from the appropriate pins of the Pi Zero W which is referenced in Figure 5.1. A simple iterating loop was created with a cyclical voltage output to allow measurement over time as shown in Figure D.11 below.



```
const pwm = require('raspi-soft-pwm');
const sleep = require('sleep');

for  (x = 0; x<100; x++) {
    raspi.init(() => {
        const output = new pwm.SoftPWM('P1-32');
        output.write(1); //100% duty cycle ~ 2.5 VDC
        console.log('PWM outputting');
        sleep.sleep(3);
        output.write(0.75); //75% duty cycle ~ 2.5 VDC
        sleep.sleep(3);
        output.write(0.5); //50% dut
        sleep.sleep(3);
        output.write(0.25); //25% du
        sleep.sleep(3);
        output.write(0); //0% duty c
        sleep.sleep(3);
        output.write(0.25); //25% du
        sleep.sleep(3);
        output.write(0.5); //50% dut
        sleep.sleep(3);
        output.write(0.75); //75% du
        sleep.sleep(3);
        output.write(1); //100% duty
        sleep.sleep(3);
    })
```

```
File /home/zero1/Desktop/pi/pwm.js saved.
File /home/zero1/Desktop/pi/pwm.js saved.
File /home/zero1/Desktop/pi/pwm.js saved.
File /home/zero1/Desktop/pi/pwm.js saved.
```

```
^C2024-08-18 20:26:30 sigHandler: Unhandled signal 2, terminating

zero1@raspberrypi:~/Desktop/pi $ sudo node pwm.js
PWM outputting
PWM outputting
PWM outputting
PWM outputting
PWM outputting
PWM outputting
PWM outputting
PWM outputting
PWM outputting
PWM outputting
PWM outputting
PWM outputting
PWM outputting
PWM outputting
PWM outputting
PWM outputting
^C2024-08-18 20:28:56 sigHandler: Unhandled signal 2, terminating

zero1@raspberrypi:~/Desktop/pi $ sudo node pwm.js
PWM outputting
```

Figure D.11: PWM signal on Picoscope during output testing.

As explained in the Results Sub-Section 5.3.6, the PWM signal from the MOSFET interface board did not produce an accurate or satisfactory output. As further work was not progressed the development circuit and code is provided up to this point of testing. The wiring diagram is shown in Figure D.13 with the programmed code also listed in Sub-Section D.4.26.

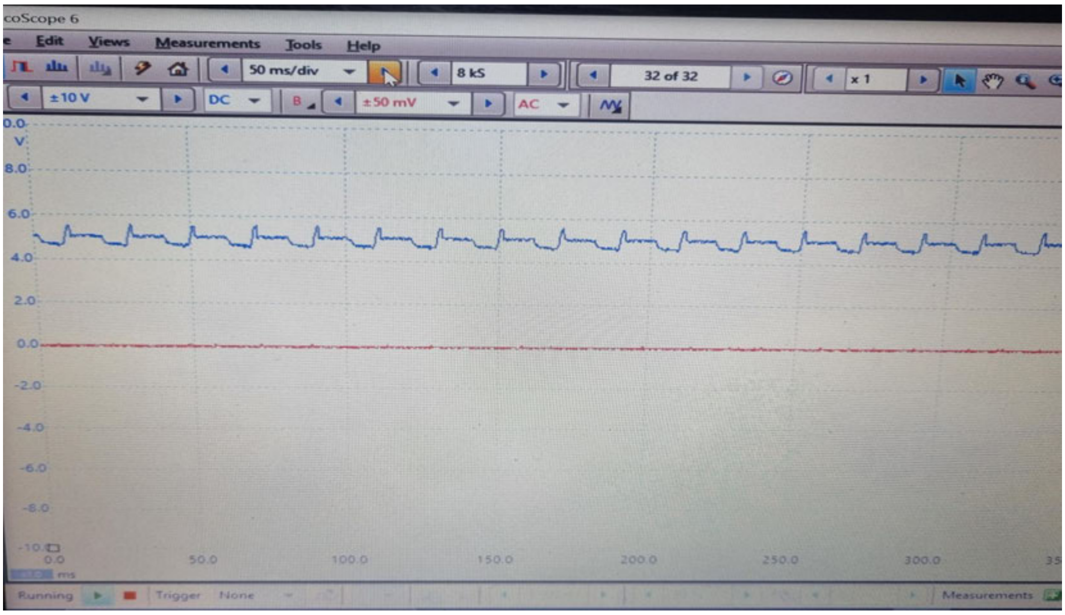


Figure D.12: Unsatisfactory PWM signal displayed on Picoscope.

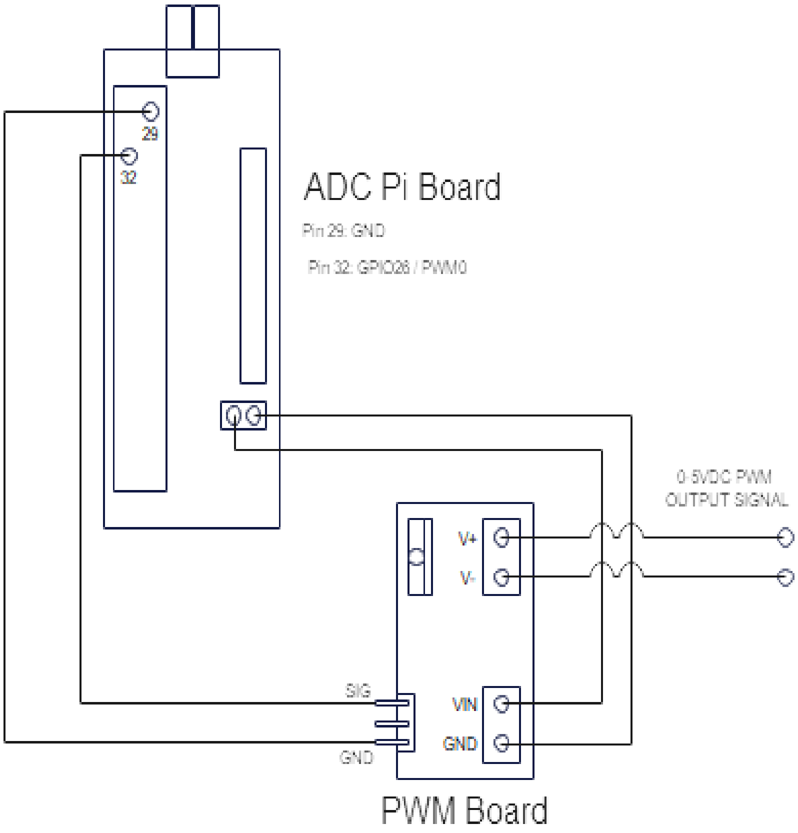


Figure D.13: PWM board wiring diagram.

D.4 Software Code Files

D.4.1 server.js code

```
//--------------- Main Central Server on Pi Five ---------------//

//--------------- Main npm dependency imports ---------------//
// Imports express for use within the node application
const express = require('express');
// Imported path library for use with system directories as required
const path = require('path');
// Imports javascript websocket library
const WebSocket = require('ws');
// Body parser for parsing form data between application states
const bodyParser = require('body-parser');
//---------------------------------------------------------//

//--------------- Custom dependency imports ---------------//
// Import time functions from the date.js module
const currentTime = require('./date.js');
// Import watchdog functions from the watchdog.js module
const wd = require('./watchdog.js');
// Import system Information
const sysInfo = require('./system.js');
// Import format IO write to DB function from serverFormatdata.js module
const format = require('./devices/serverFormatdata.js');
// Import websockets functions from the socket.js module
const wskt = require('./serverSocket.js');
// Import mysql database function from the database.js module
const db = require('./database.js');
//---------------------------------------------------------//

//--------------- Express application created -------------//
/* Assigns the JavaScript express function which is the request function
   handler to be passed to the NodeJS HTTP server. */
   const app = express();
   // Creates a HTTP server instance and uses the express handler within.
   const server = require('http').createServer(app);
//---------------------------------------------------------//

//---------------- Main HMI - MVC imports ----------------//
// The below command tells express that EJS is the Template Engine of choice for Project
app.set('view engine', 'ejs');
// Set route handler constants for routing app / data to HMTL HMI screens
const hmiroutes = require('./routes/hmi.js');
const errorController = require('./controllers/error.js');
// Used for allowing url encoding within the app
app.use(bodyParser.urlencoded({extended: false}));
// Below is used to direct app to the static public directory for serving images/css etc to page
requests
app.use(express.static(path.join(__dirname, '/public')));
// Below commands run the route handlers set above
```

```
app.use(hmiroutes);
app.use(errorController.get404);
//-------------------------------------------------------//

//------------- Date Time Logging for Server ------------//
// Globally scoped variable definitions
// These are used in a setInterval but need to be global so are declared here
var dateTime;
let GMT;
// This runs the timeStamp function imported from date.js to display current time within server.js
// Takes the current time every 500ms
setInterval (function() {
    let dateTimeGMT = currentTime.timeStamp();
    dateTime = dateTimeGMT[0].toString();
    GMT = dateTimeGMT[1].toString();
},100)
//-------------------------------------------------------//

//------------- System watchdog testing here ------------//
wd.sysWatchdog();
wd.sysWatchdogRefresh();

//------------------------- HTTP Server -------------------------//
// The IP can be commented out to allow localhost testing of comms on laptop etc.
server.listen(3000, '192.168.0.4', () => {
    console.log("Server up, listening on port: 3000")
});

//----- Below section for functions relating to WS operation ----//
const wss = new WebSocket.Server({ server:server });
//const ws = new WebSocket('ws://192.168.0.4:3000'); //******* get rid of this and ws below */
const socket = wskt.wsServer(wss);

//------------------ mySQL Database connection ------------------//
const sqlConn = db.myDB();

// Logs system info into DB every 10 seconds
setInterval (function() {
    let sysData = sysInfo.createsysMap();
    let formattedData = format.formatDBdata(sysData);
    db.sysInfoToDB ('serversysdata', formattedData.keyArray, formattedData.valueArray, dateTime);
},10000)
//------------------ End of Server Application ------------------//
```

D.4.2 client1.js code

```
//************** Client code for Raspberry Pi Zero W unit 1 **************//

//************** Below section for dependencies **************//
const express = require('express');
const app = express();
const server = require('http').createServer(app);
const WebSocket = require('ws');
// Performance timing for testing and monitoring code
const { performance } = require('perf_hooks');
// Import time functions from the date.js module
const currentTime = require('./date.js');
// Import time functions from the client1iodata.js module
const c1data = require('./devices/client1iodata.js');
// Import websockets functions from the socket.js module
const wskt = require('./clientSocket.js');
// Import system Information
const sysInfo = require('./system.js');
//**********************************************************************//

// Globally scoped variable definitions
// These are used in a setInterval but need to be global so are declared here
var dateTime;

// This runs the timeStamp function imported from date.js to display current time within server.js
// Takes the current time every 100ms
setInterval (function() {
    let dateTimeGMT = currentTime.timeStamp();
    dateTime = dateTimeGMT[0].toString();
},100)

// Display system Info to the console
let systemInformation = sysInfo.displaySystemInfo();
//console.log('System info output:', systemInformation);

// Log the execution time for Client operations
const start = performance.now();
// Connect to the WS piFive server here
var clientNumber = '1'
console.log('Starting WS now');
// The below is used for network, commented is for single device testing
const socket = new WebSocket('ws://192.168.0.4:3000' /* 'ws://localhost:3000' */);

// If Server side is running the below can be executed
const clientSocket = wskt.wsClient(socket, clientNumber, c1data.updateIO);

const end = performance.now();
console.log(`Time taken to execute the websocket function is ${end - start}ms.`);
```

```
setInterval(() => {
    console.log('Client running and time is: ', dateTime);
}, 5000);
```

D.4.3 client2.js code

```
//************** Client code for Raspberry Pi Zero W unit 2 **************//

//************** Below section for dependencies **************//
const express = require('express');
const app = express();
const server = require('http').createServer(app);
const WebSocket = require('ws');
// Performance timing for testing and monitoring code
const { performance } = require('perf_hooks');
// Import time functions from the date.js module
const currentTime = require('./date.js');
// Import time functions from the client1iodata.js module
const c2data = require('./devices/client2iodata.js');
// Import websockets functions from the socket.js module
const wskt = require('./clientSocket.js');
// Import system Information
const sysInfo = require('./system.js');
//**********************************************************************//

// Globally scoped variable definitions
// These are used in a setInterval but need to be global so are declared here
var dateTime;

// This runs the timeStamp function imported from date.js to display current time within server.js
// Takes the current time every 100ms
setInterval (function() {
    let dateTimeGMT = currentTime.timeStamp();
    dateTime = dateTimeGMT[0].toString();
},100)

// Display system Info to the console
let systemInformation = sysInfo.displaySystemInfo();

// Connect to the WS piFive server here
var clientNumber = '2'
console.log('Starting WS now');
// The below code connects to pi five on network, use commented code if local device testing only
const socket = new WebSocket( 'ws://192.168.0.4:3000' /* 'ws://localhost:3000' */ );
const clientSocket = wskt.wsClient(socket, clientNumber, c2data.updateIO);

setInterval(() => {
    console.log('Client running and time is: ', dateTime);
}, 5000);
```

D.4.4 serverSocket.js code

```
// Function for Server WebSockets connectivity

// Import watchdog functions from the watchdog.js module
const wd = require('./watchdog.js');
// Import io write to DB function from database.js module
const sqlDB = require('./database.js');
// Import format IO write to DB function from serverFormatdata.js module
const format = require('./devices/serverFormatdata.js');
// Import time functions from the date.js module
const currentTime = require('./date.js');

var wss;
var serverTime;

function wsServer(wss) {

    wss.on('connection', function connection(wss) {

        // Below code always executed when new connection established
        wd.serverWatchdogInitiate(wss);
        console.info('New Client connected');

        // Listen for messages
        wss.addEventListener('message', function (e){msg(e)})

        function msg(e) {
            let messageData = JSON.parse(e.data);

            // If msg data type is for Watchdog Intitiate
            if (messageData.type === 'wdinit') {
                let clientID = messageData.clientID;
                //*********** Watchdog Timer runs here ************//
                wd.serverCommsWatchdogTimeout(wss, clientID);
            }
            // If msg data type is for Watchdog Run
            else if (messageData.type === 'wdrun') {
                let clientID = messageData.clientID;
                wd.serverCommsWatchdogRefresh(clientID);

            }
            // If msg data type is for Input / Output Logging
            else if (messageData.type === 'io') {
                // Log the local server time to allow for latency checks etc
                let dateTimeGMT = currentTime.timeStamp();
                serverTime = JSON.stringify(dateTimeGMT[0]);//.toString();
                let formattedData = format.formatDBdata(messageData.data);
                // Check which client and write to according mySQL table
                if (messageData.clientID === '1') {
                    sqlDB.ioToDB('client1iodata', formattedData.keyArray, formattedData.valueArray,
serverTime, messageData.time);
                }
                else if (messageData.clientID === '2') {
                    sqlDB.ioToDB('client2iodata', formattedData.keyArray, formattedData.valueArray,
serverTime, messageData.time);
                }
            }
            // If msg data type is for Alarm Management
            else if (messageData.type === 'alarm') {
                let formattedData = format.formatDBdata(messageData.data);
                sqlDB.alarmToDB('alarmdata', formattedData.keyArray, formattedData.valueArray,
messageData.time, messageData.clientID);
            }
            // If msg data type is for System Information Logging
            else if (messageData.type === 'sys') {
                let clientID = messageData.clientID;
                let formattedData = format.formatDBdata(messageData.data);

                if (messageData.clientID === '1') {
                    console.log(clientID, 'Client system message : ', messageData);
                    sqlDB.sysInfoToDB ('client1sysdata', formattedData.keyArray,
formattedData.valueArray, messageData.time);
                }
                else if (messageData.clientID === '2') {
                    console.log(clientID, 'Client system message : ', messageData);
                    sqlDB.sysInfoToDB ('client2sysdata', formattedData.keyArray,
formattedData.valueArray, messageData.time);
                }
            }
            // If msg data type is for any other non-specified types
            else {
                console.log('Client message : ', messageData);
            }
        }

        wss.on('close', function(event) {
            if (event.wasClean) {
                console.log(`[close] Connection closed cleanly, code=${event.code}
reason=${event.reason}`);
                } else {
                console.log('[close] Connection died');
                }
            // connection closed, discard old websocket and create a new one in 5s

        });

        wss.on('error', function(event) {
            console.log('error with WS connection')
            // can add error logs to DB if reqd
        });
    });

}

module.exports = {
    wsServer
};
```

D.4.5 clientSocket.js code

```
// Function for websocket connectivity

// Import watchdog functions from the watchdog.js module
const wd = require('./watchdog.js');
// Import time functions from the date.js module
const currentTime = require('./date.js');
// Import sendData function from the clientSendData.js module
const clientData = require('./devices/clientSendData.js');
// Import system information
const sysInfo = require('./system.js');

// Globally scoped variable definitions
var WDflag;

// Below code triggers flag for watchdog after first server initialise command.
if (WDInit = false) {
    WDflag = false;
}
else if (WDinit = true) {
    WDflag = true;
}

// Main WebSocket Server for export to clients to use within individual applications.

    // If Server side is running the below can be executed
    function wsClient(socket, clientNumber, dataFunction) {
        try {

            // Connection opened event listener
            socket.addEventListener('open', function (event) {
                // Monitor the node app execution time overall and output to console
                //const start = performance.now();
                //const end = performance.now();
                //console.log(`Time taken to execute the WebSockets function is ${end - start}ms.`);
                console.log(clientNumber, 'connected to Pi Five WS Server');

                // Interval terminal comms to display system and operating values every 10 seconds
                const id = setInterval(function () {
                    socket.send(JSON.stringify(process.memoryUsage()), function () {
                    });
                }, 10000);
            });

            // Listen for messages
            socket.addEventListener('message', function (e){msg(e)});

            // Function handler for message events within Socket operations
            function msg(e) {
                let messageData = JSON.parse(e.data);

                if(messageData.type === 'wd' && messageData.ID === 'server') {

                    // Watchdog Intialisation step - Send back the clientNumber to Server to create
new watchdog timeout
                    socket.send(JSON.stringify({type: 'wdinit', clientID: clientNumber}));
                    console.log('Message sent from: ', messageData.ID)
                    // Initiate Watchdog flag which sets watchdog within clientCommsWatchdogSend to
TRUE (DECLARED AT TOP OF PAGE)
                    WDinit = true;
                }
                // Display any other message data apart from watchdog messages
                else {
                    console.log('Server message : ', messageData);
                }
            }

            // Watchdog periodic reset function, sends reset to Server once intitiated above by Server
            wd.clientCommsWatchdogSend(WDflag, socket, clientNumber);

            // Client I/O data is sent using this function
            clientData.sendData(clientNumber, socket, dataFunction);



            // System data periodically sent to pi Five DB for data logging
            sysInfo.sendSystemData(clientNumber, socket,sysInfo.createsysMap);

            // Closed socket event listener
            socket.addEventListener('close', function(event) {
                if (event.wasClean) {
                    console.log(`Connection closed cleanly, code=${event.code}
reason=${event.reason}`);
                    }
                else {
                    console.log('Connection has died');
                    }
                });

            // Error event listener
            socket.addEventListener('error', console.log("Websockets error has occured"));
            }
        catch (e) {
            // Logs the cause of error to user
            console.log('Pi Five Server Offline, Please start first');
        }
};
```

```
module.exports = {
    wsClient
}
```

D.4.6 watchdog.js code

```
//----------- Section used for watchdog function definitions -----------//

// Import time functions from the date.js module
const currentTime = require('./date.js');

///////////////// System watchdog set for 1 second timeout /////////////
// Global var for functions and exports
let watchdogTimer;
let commsTimer = [];
var clientID;

function sysWatchdog() {
    watchdogTimer = setTimeout(() => {
    console.error('System Watchdog timer elapsed, Runtime error');
    process.exit(1);
  }, 1000);
}

function sysWatchdogRefresh() {
    setInterval(() => {
    clearTimeout(watchdogTimer);
    }, 500);
}

///////////////////////////// Comms watchdog ///////////////////////////

/* First message sent to clients upon connection to start watchdog comms
(passes the socket so it can use send function etc) */
function serverWatchdogInitiate(socket) {
    let startMsg = JSON.stringify({type: 'wd', ID: 'server'});
    console.log('Server comms watchdog initiated with new connected client');
    socket.send(startMsg);
}

// If 5000ms elapses without an update then ws.terminate, reoccuring interval timer
function serverCommsWatchdogTimeout(socket, clientID) {
    console.log('Client', clientID, 'timeout started');
    let clientName = clientID.toString();
    let ID = Number(clientID);
    console.log('Client name is:', clientName);
    commsTimer[ID] = setInterval(() => {
        console.error(clientName, 'Comms Watchdog Timer Elapsed, Comms Error');
        socket.terminate();
    }, 500)};

// Stops the watchdog interval timer and starts again
function serverCommsWatchdogRefresh(clientID) {
    let clientName = clientID.toString();
    let ID = Number(clientID);
    let dateTimeGMT = currentTime.timeStamp();
    dateTime = dateTimeGMT[0].toString();
    // method to check if watchdog reset running in terminal
    console.log(clientName, 'Comms Watchdog reset at time: ', dateTime);
    clearInterval(commsTimer[ID]);
};

// Sends a char flag 'wdrun' every 4000mS to the server.
// If not rcvd by server WS msg, comms have issue so allow timeout to close connection
function clientCommsWatchdogUpdate(WDflag, socket, clientNumber) {
    if (WDflag == true) {
        let sendMsg = JSON.stringify({ type: 'wdrun', clientID:clientNumber});
        let dateTimeGMT = currentTime.timeStamp();
        dateTime = dateTimeGMT[0].toString();
        console.log('Client',clientNumber, 'Sending WD reset back to Server at time:',
dateTime);
        socket.send(sendMsg);
    }
    else {
        console.log('WD not running');
    }
};

function clientCommsWatchdogSend(WDflag, socket, clientNumber) { setInterval(() =>
clientCommsWatchdogUpdate(WDflag, socket, clientNumber), 500)};    // Polls for new data
every 500mS and returns to server

module.exports = {
    watchdogTimer,
    commsTimer,
    sysWatchdog,
    sysWatchdogRefresh,
    serverWatchdogInitiate,
    serverCommsWatchdogTimeout,
    serverCommsWatchdogRefresh,
    clientCommsWatchdogUpdate,
    clientCommsWatchdogSend
  };
```

D.4.7 system.js code

```
// Import time functions from the date.js module
const currentTime = require('./date.js');
// Import system information monitor
const os = require('os');
// Allocating process module
const process = require('process');
//-----------------------------------------------------------------------------------------
-------

// Can be used to display system info for the running instance on console etc.
// Runs at startup of each client
function displaySystemInfo() {
    let freeMemory = os.freemem() / 1024 / 1024; // 1024/1024 converts to MB
    console.log("Available free memory:", freeMemory.toFixed(2), "MB");
    let cpuCores = os.cpus().length;
    console.log("The number of CPU cores is:", cpuCores);
    // Calling process.cpuUsage() method
    let usage = process.cpuUsage();
    // Printing returned value
    console.log('CPU usage is:', usage.user, usage.system);
}
//-----------------------------------------------------------------------------------------
--------

/* Creates system info map for server.js to use directly and also to be
   passed into below 'sysInfo' client wbesocket send function  */
function createsysMap (){
    // System values generated below
    let freeMemory = os.freemem() / 1024 / 1024; // 1024/1024 converts to MB
    freeMemory = freeMemory.toFixed(2);
    let usage = process.cpuUsage();
    let userUsage = usage.user;
    let sysUsage = usage.system;

    sysInfoDataMap = new Map();
    sysInfoDataMap.set('free memory',freeMemory);
    sysInfoDataMap.set('cpu user usage',userUsage);
    sysInfoDataMap.set('cpu system usage',sysUsage);

    // Converts the created map to JSON for websocket transmit and DB write
    const obj = Object.fromEntries(sysInfoDataMap);
    const mysysInfo = JSON.stringify(obj);
    return(mysysInfo);
};

// Used to send system info to mysql DB for logging
function sysInfo (clientID, socket, createsysMap){
    // Assigns sys info returned by 'createsysMap above to var
    var sysMap = createsysMap();
    console.log('system info is:',sysMap);
    // Add timestamp data
    let dateTimeGMT = currentTime.timeStamp();
    dateTime = dateTimeGMT[0].toString();
    // The data read and returned by the above is then sent using WS to Server
    if (socket.bufferedAmount == 0) {
        socket.send(JSON.stringify({type: 'sys', clientID: clientID, time: dateTime, data:
sysMap}), function () {
        });
    } else {
        console.log('Websockets buffer full');
    }
};

// The below generic function calls the above at the set interval (This is exported to
clientSocket.js)
function sendSystemData(clientID, socket, createsysMap) { setInterval(() =>
sysInfo(clientID, socket, createsysMap), 10000)};

module.exports = {
    displaySystemInfo,  // exported to client1/2.js
    createsysMap,       // exported to clientSocket.js & server.js
    sendSystemData      // exported to clientSocket.js
  };
```

D.4.8 date.js code

```
// Date module created for timestamping operations

function timeStamp() {

    let d = new Date();                         // Grabs a new date instance from
node date module
    let m = d.getMilliseconds();                // Takes a separate millisecond
value (not in first call)
    let milli = m.toString();                   // Converts to string
    let addMilli = (':' + milli +' ')           // Creates a string formatted with
ms to add to date string
    let myTime = d.toString();                  // Converts to string
    let splitTime = myTime.split(' (');         // Removes the (AWST)
    let section1 = splitTime[0].length;         // Adds millisecs and returns
separate time and GMT offset as below
    let newDate = splitTime[0].slice(0,(section1-9)) + addMilli +
splitTime[0].slice((section1-8),section1);
    let finalDate = newDate.slice(0,(newDate.length-9));
    let GMT = newDate.slice((newDate.length-8),newDate.length)

    return [finalDate,GMT];
}

module.exports = {
    timeStamp
  };
```

D.4.9 database.js code

```
// Database logic for export to the main node Pi Five Server

// For DB creation run the code with server.js once to create the DB then null the NOTE #1
code
// and activate the NOTE #2 code which points to the created DB and allows operations etc.

// Import dependencies
// Below modified to allow encrypted authentication compatible with nodeJS
const mysql = require('mysql2');
var connection;

/* Below is created as a function not a class as only a single instance is required between
   the Pi Five App and DB. If connecting multiple clients to DB then a class with static
   initialisation block should be used. */
function myDB() {
    connection = mysql.createConnection({
        host: 'localhost',
        port: '3306',
        user: 'scott',
        password: 'scottpw',
        database: 'piFiveDB' // NOTE #2 enable this line after first run and DB/tables
created //
    });

    connection.connect((err) => {
        if (err) {
            throw err;
        }
        else {
            console.log('Connected to MySql Server!');
            /* NOTE #1 // Once initialised the below lines of code is redundant and
               replaced with // NOTE #2 // code above, commented out

            connection.query('create database piFiveDB',function (err, result) {
                if(err) {
                    console.log(err);
                }
                else {
                    console.log('Database created');
                }
            })
            */

            // Once created the below lines of code are redundant
            var ioTable2 = "CREATE TABLE client1iodata (ID int NOT NULL AUTO_INCREMENT, tag
VARCHAR(255), value VARCHAR(255), servertime VARCHAR(255),clienttime VARCHAR(255), PRIMARY
KEY (ID))";
            connection.query(ioTable2, function (err, result) {
                if(err) {
                    console.log(err);
                }
                else {
                    console.log('Table created')
                }
            })
            var ioTable3 = "CREATE TABLE client2iodata (ID int NOT NULL AUTO_INCREMENT, tag VARCHAR(255), value VARCHAR(255), servertime VARCHAR(255), clienttime VARCHAR(255), PRIMARY KEY (ID))";
            connection.query(ioTable3, function (err, result) {
                if(err) {
                    console.log(err);
                }
                else {
                    console.log('Table created')
                }
            })
            var sysTable1 = "CREATE TABLE client1sysdata (ID int NOT NULL AUTO_INCREMENT, tag VARCHAR(255), value VARCHAR(255), clienttime VARCHAR(255), PRIMARY KEY (ID))";
            connection.query(sysTable1, function (err, result) {
                if(err) {
                    console.log(err);
                }
                else {
                    console.log('Table created')
                }
            })
            var sysTable2 = "CREATE TABLE client2sysdata (ID int NOT NULL AUTO_INCREMENT, tag VARCHAR(255), value VARCHAR(255), clienttime VARCHAR(255), PRIMARY KEY (ID))";
            connection.query(sysTable2, function (err, result) {
                if(err) {
                    console.log(err);
                }
                else {
                    console.log('Table created')
                }
            })
            var sysTable3 = "CREATE TABLE serversysdata (ID int NOT NULL AUTO_INCREMENT, tag VARCHAR(255), value VARCHAR(255), clienttime VARCHAR(255), PRIMARY KEY (ID))";
            connection.query(sysTable3, function (err, result) {
                if(err) {
                    console.log(err);
                }
                else {
                    console.log('Table created')
                }
            })
            var alarmTable = "CREATE TABLE alarmdata (ID int NOT NULL AUTO_INCREMENT, tag
VARCHAR(255), status VARCHAR(255), clienttime VARCHAR(255), clientnumber VARCHAR(255),
PRIMARY KEY (ID))";
            connection.query(alarmTable, function (err, result) {
                if(err) {
                    console.log(err);
                }
                else {
                    console.log('Table created')
                }
            })
        }
    });
    return(connection);
};

// function for writing IO data from clients into database
function ioToDB (tableName, tags, values, servertime, datetime) {
    datetime = JSON.stringify(datetime);
    for( j=0; j<tags.length; j++) {
        let db = "INSERT INTO " + tableName + " (tag, value, servertime, clienttime) \
        VALUES("+ tags[j] +","+ values[j] +", "+ servertime +","+ datetime +")";
        connection.query(db, function(err, result) {
            if(err) throw err;
        })
    }
    console.log('Updated client I/O data inserted');
};

// Function for reading device value from database to HMI
var myval;
function ioDataFromDB () {
        let db = "select * from client1iodata WHERE tag = 'PIT-210' ORDER BY ID DESC LIMIT
1";
        connection.query(db, function(err, result) {
            if(err) throw err;
            myval = result[0].value;
            console.log('************ Device data pulled from database', myval);
        })
        return(myval);
};

// Function for writing alarm data from clients into database
function alarmToDB (tableName, tags, values, datetime, clientID) {
    datetime = JSON.stringify(datetime);
    clientID = JSON.stringify(clientID);
    for( j=0; j<tags.length; j++) {
        let db = "INSERT INTO " + tableName + " (tag, status, clienttime, clientnumber)
VALUES("+ tags[j] +","+ values[j] +","+ datetime +","+ clientID +")";
        connection.query(db, function(err, result) {
            if(err) throw err;
        })
    }
    console.log('Updated client alarm data inserted');
};

// Function for writing system info data from server and clients in database
function sysInfoToDB (tableName, tags, values, datetime) {
    datetime = JSON.stringify(datetime);
    for( j=0; j<tags.length; j++) {
        let db = "INSERT INTO " + tableName + " (tag, value, clienttime) VALUES("+ tags[j]
+","+ values[j] +","+ datetime +")";
        connection.query(db, function(err, result) {
            if(err) throw err;
        })
    }
    console.log('Updated system data inserted');
};

module.exports = {
    myDB,
    ioToDB,
    alarmToDB,
    sysInfoToDB,
    ioDataFromDB,
    myval,
    connection
};
```

D.4.10 devices/client1iodata.js code

```
// Import device function from the clientDigital.js module
const digitalIO = require('./clientDigital.js');
// Import device class and functions from the clientAnalog.js module
const analogIO = require('./clientAnalog.js');
// Initialises any simulated IO values for below function (only for testing)
const initVar = (function () {
    count = 0;
    ioVar1 = 1;     // Used to initiate simulate Discrete values
    ioVar2 = 0;
    ioVar3 = 1;
    ioVar4 = 0;
    ioVar5 = 0;
    ioVar6 = 0;
    ioVar7 = 0;
    ioVar8 = 0;
    ioVar9 = 0;
    ioVar10 = 0;
    ioVar11 = 0;
    ioVar12 = 0;
    ioVar13 = 0;
    ioVar14 = 0;
    ioVar15 = 0;
    ioVar16 = 0;
    alarmFlag = false;      // Used as one-shot for alarm sending to database - Until
resetAlarm() method called for aiDevice()
    transmitFlag = false;   // Used as above to send the ws data only once - passed through
with the message data
    ai1Voltage = 1;         // Used to initiate simulate Analog voltage
    ai2Voltage = 1.2;       // Used to initiate simulate Analog voltage
    ai3Voltage = 3.5;       // Used to initiate simulate Analog voltage
    ai4Voltage = 2.6;       // Used to initiate simulate Analog voltage
    return function() {
        return count, ioVar1, ioVar2, ioVar3, ioVar4, ioVar5, ioVar6, ioVar7, ioVar8,
        ioVar9, ioVar10, ioVar11, ioVar12, ioVar13, ioVar14, ioVar15, ioVar16,
        ai1Voltage, ai2Voltage, ai3Voltage, ai4Voltage }
})();

var aiDevice1, aiDevice2, aiDevice3, aiDevice4;

/* Updates IO created in the above function for recording 'simulated' values in Database.
   When runnning without simulated values on units, the internal ADC Pi and GPIO functions
   will pull the IO interface data to send with this function used within clientSocket.js
*/

/* Below function is for reading analog and discrete IO values and updating IO objects to
be
sent to Pi Five Server over Websocket function (exported to clientSocket.js) */
function updateIO() {
    //------------------------------ For Simulation & Test Only ---------------------------
----------//
    /* Below if/else used to toggle values and increment analog signal values for testing
back to database,
       to be removed when running real I/O inputs */
    if (count == 1) {
        count = 0;
        ioVar1 = 1, ioVar2 = 0, ioVar3 = 1, ioVar4 = 1, ioVar5 = 1, ioVar6 = 0, ioVar7 = 1,
ioVar8 = 1,
        ioVar9 = 1, ioVar10 = 0, ioVar11 = 1, ioVar12 = 1,ioVar13 = 1, ioVar14 = 0, ioVar15
= 1, ioVar16 = 1,
        ai1Voltage = ai1Voltage + 0.00005, ai2Voltage = ai2Voltage + 0.00005, ai3Voltage =
ai3Voltage + 0.00005, ai4Voltage = ai4Voltage + 0.00005; // Increases voltage for testing
alarms each scan
    }
    else {
        count = 1;
        ioVar1 = 0, ioVar2 = 1, ioVar3 = 0, ioVar4 = 0, ioVar5 = 0, ioVar6 = 1, ioVar7 = 0,
ioVar8 = 0,
        ioVar9 = 0, ioVar10 = 1, ioVar11 = 0, ioVar12 = 0,ioVar13 = 0, ioVar14 = 1, ioVar15
= 0, ioVar16 = 0,
        ai1Voltage = ai1Voltage + 0.00005, ai2Voltage = ai2Voltage + 0.00005, ai3Voltage =
ai3Voltage + 0.00005, ai4Voltage = ai4Voltage + 0.00005; // Increases voltage for testing
alarms each scan
        }
    //------------------------------ End of Simulation & Test code ------------------------
------------//

    // Creates object instances of the discrete IO devices with their current measured
values
    let ioData1 = new digitalIO.ioDevice('XI-1001', ioVar1);
    let ioData2 = new digitalIO.ioDevice('XI-1002', ioVar2);
    let ioData3 = new digitalIO.ioDevice('XI-1003', ioVar3);
    let ioData4 = new digitalIO.ioDevice('XI-1004', ioVar4);
    let ioData5 = new digitalIO.ioDevice('XI-1005', ioVar5);
    let ioData6 = new digitalIO.ioDevice('XI-1006', ioVar6);
    let ioData7 = new digitalIO.ioDevice('XI-1007', ioVar7);
    let ioData8 = new digitalIO.ioDevice('XI-1008', ioVar8);
    let ioData9 = new digitalIO.ioDevice('XI-1009', ioVar9);
    let ioData10 = new digitalIO.ioDevice('XI-1010', ioVar10);
    let ioData11 = new digitalIO.ioDevice('XI-1011', ioVar11);
    let ioData12 = new digitalIO.ioDevice('XI-1012', ioVar12);
    let ioData13 = new digitalIO.ioDevice('XI-1013', ioVar13);
    let ioData14 = new digitalIO.ioDevice('XI-1014', ioVar14);
    let ioData15 = new digitalIO.ioDevice('XI-1015', ioVar15);
    let ioData16 = new digitalIO.ioDevice('XI-1016', ioVar16);
    // Creates analog device (object) using class constructor
    aiDevice1 = new analogIO.aiDevice('PIT-210', ai1Voltage, 0, 2000, 'kpa', 800, 0, 0, 0,
4000, 0, 0, 0, 5, 0);
    aiDevice2 = new analogIO.aiDevice('PIT-460', ai2Voltage, 0, 300, 'kpa', 0, 0, 0, 0,
4000, 0, 0, 0, 5, 0);
    aiDevice3 = new analogIO.aiDevice('TIT-320', ai3Voltage, 0, 100, 'deg C', 0, 0, 0, 0,
0, 0, 0, 0, 5, 0);
    aiDevice4 = new analogIO.aiDevice('FIT-500', ai4Voltage, 0, 150, 'deg C', 0, 0, 0, 0,
0, 0, 0, 0, 5, 0);
    // Returns Scaled PV for sending to Database in above scan, scanAnalog not required
here therefore

    // Create a new Data Map of all discrete and analog I/O TAG and raw values instances
for database transmit
    var ioDataMap = new Map();
    ioDataMap.set(ioData1.tag, ioData1.raw);
    ioDataMap.set(ioData2.tag, ioData2.raw);
    ioDataMap.set(ioData3.tag, ioData3.raw);
    ioDataMap.set(ioData4.tag, ioData4.raw);
    ioDataMap.set(ioData5.tag, ioData5.raw);
    ioDataMap.set(ioData6.tag, ioData6.raw);
    ioDataMap.set(ioData7.tag, ioData7.raw);
    ioDataMap.set(ioData8.tag, ioData8.raw);
    ioDataMap.set(ioData9.tag, ioData9.raw);
    ioDataMap.set(ioData10.tag, ioData10.raw);
    ioDataMap.set(ioData11.tag, ioData11.raw);
    ioDataMap.set(ioData12.tag, ioData12.raw);
    ioDataMap.set(ioData13.tag, ioData13.raw);
    ioDataMap.set(ioData14.tag, ioData14.raw);
    ioDataMap.set(ioData15.tag, ioData15.raw);
    ioDataMap.set(ioData16.tag, ioData16.raw);
    ioDataMap.set(aiDevice1.tag, aiDevice1.scaledPV.toFixed(1));
    ioDataMap.set(aiDevice2.tag, aiDevice2.scaledPV.toFixed(1));
    ioDataMap.set(aiDevice3.tag, aiDevice3.scaledPV.toFixed(1));
    ioDataMap.set(aiDevice4.tag, aiDevice4.scaledPV.toFixed(1));

    // Checking alarm activation of devices - Only 1 Analog Device being tested currently
    var aiDev1param = aiDevice1.checkAlarms();
    // Below console output used for testing and visibility only
    // console.log('Scaled PV is: ', aiDev1param[0].toFixed(0), ' | Alarm Flag is: ',
aiDev1param[1], ' | Alarm Status is: ', aiDev1param[2], ' | Device Tag is: ',
aiDev1param[3]);

    // Create Alarm data if alarmFlag generated in above alarmCheck() method call for each
device
    var alarmDataMap = new Map();
    if (aiDev1param[1] == 'true' && alarmFlag == false) {
        alarmDataMap.set(aiDevice1.tag, aiDev1param[2]);
        alarmFlag = true;
        transmitFlag = true;
    }
    else if (aiDev1param[1] == 'false' && alarmFlag == true) {
        /* If aiDevice1.resetAlarm() method is called the aiDev1param[1] is set false and
           this 'else if' will run resetting the logging function to DB */
        alarmFlag = false;
    }
    else {
        // Creates transmit one-shot, logs nothing to Database until the alarm is reset
        transmitFlag = false;
    }

    // Converts the created data map of analog and discrete tag/values to JSON for
websocket transmit
    const obj1 = Object.fromEntries(ioDataMap);
    const myioData = JSON.stringify(obj1);
    // Converts the created data map of alarm tag/values to JSON for websocket transmit
    const obj2 = Object.fromEntries(alarmDataMap);
    const myAlarmData = JSON.stringify(obj2);
    const wsflag = JSON.stringify(transmitFlag);

    return([myioData, myAlarmData, wsflag]);
};

module.exports = {
    updateIO, // used within client/1/2.js
    aiDevice1

};
```

D.4.11 devices/client2iodata.js code

```
// Import device function from the clientDigital.js module
const digitalIO = require('./clientDigital.js');
// Import device class and functions from the clientAnalog.js module
const analogIO = require('./clientAnalog.js');
// Initialises any simulated IO values for below function (only for testing)
const initVar = (function () {
    count = 0;
    ioVar17 = 1;     // Used to initiate simulate Discrete values
    ioVar18 = 0;
    ioVar19 = 1;
    ioVar20 = 0;
    ioVar21 = 1;
    ioVar22 = 1;
    ioVar23 = 0;
    ioVar24 = 1;
    ioVar25 = 0;
    ioVar26 = 0;
    ioVar27 = 1;
    ioVar28 = 0;
    ioVar29 = 1;
    ioVar30 = 1;
    ioVar31 = 1;
    ioVar32 = 0;
    alarmFlag = false;      // Used as one-shot for alarm sending to database - Until
resetAlarm() method called for aiDevice()
    transmitFlag = false;   // Used as above to send the ws data only once - passed through
with the message data
    ai5Voltage = 1.6;       // Used to initiate simulate Analog voltage
    ai6Voltage = 1.02;      // Used to initiate simulate Analog voltage
    ai7Voltage = 1.01;      // Used to initiate simulate Analog voltage
    ai8Voltage = 1.07;      // Used to initiate simulate Analog voltage
    return function() {
        return count, ioVar17, ioVar18, ioVar19, ioVar20, ioVar21, ioVar22, ioVar23,
ioVar24,
        ioVar25, ioVar26, ioVar27, ioVar28, ioVar29, ioVar30, ioVar31, ioVar32,
        ai5Voltage, ai6Voltage, ai7Voltage, ai8Voltage }
})();

var aiDevice5, aiDevice6, aiDevice7, aiDevice8;

// Updates IO created in the above function for recording 'simulated' values in Database.
/* When runnning properly on units the internals of this function will pull the IO
interface data
// to send and function is used within clientSocket.js */


/* Below function is for reading analog and discrete IO values and updating IO objects to
be
sent to Pi Five Server over Websocket function (exported to clientSocket.js) */
```

```
function updateIO() {

    //----------------------------------- For Test Only -----------------------------------
--//
    /* Below if/else used top toggle values for testing back to database
       to be removed when running direct on Pi units */
    if (count == 1) {
        count = 0;
        ioVar17 = 1, ioVar18 = 0, ioVar19 = 1, ioVar20 = 1, ioVar21 = 1, ioVar22 = 0,
ioVa23 = 1, ioVar24 = 1,
        ioVar25 = 1, ioVar26 = 0, ioVar27 = 1, ioVar28 = 1,ioVar29 = 1, ioVar30 = 0,
ioVar31 = 1, ioVar32 = 1,
        ai5Voltage = ai5Voltage + 0.000015, ai6Voltage = ai6Voltage + 0.000015, ai7Voltage
= ai7Voltage + 0.000015, ai8Voltage = ai8Voltage + 0.000015;
        // Increases voltage for testing alarms each scan
    }
    else {
        count = 1;
        ioVar17 = 0, ioVar18 = 1, ioVar19 = 0, ioVar20 = 0, ioVar21 = 0, ioVar22 = 1,
ioVar23 = 0, ioVar24 = 0,
        ioVar25 = 0, ioVar26 = 1, ioVar27 = 0, ioVar28 = 0,ioVar29 = 0, ioVar30 = 1,
ioVar31 = 0, ioVar1632 = 0,
        ai5Voltage = ai5Voltage + 0.000015, ai6Voltage = ai6Voltage + 0.000015, ai7Voltage
= ai7Voltage + 0.000015, ai8Voltage = ai8Voltage + 0.000015;
        // Increases voltage for testing alarms each scan
        }
    //-------------------------------------------------------------------------------------
--//

    // Creates object instances of the discrete IO devices with their current measured
values
    let ioData17 = new digitalIO.ioDevice('XI-1017', ioVar17);
    let ioData18 = new digitalIO.ioDevice('XI-1018', ioVar18);
    let ioData19 = new digitalIO.ioDevice('XI-1019', ioVar19);
    let ioData20 = new digitalIO.ioDevice('XI-1020', ioVar20);
    let ioData21 = new digitalIO.ioDevice('XI-1021', ioVar21);
    let ioData22 = new digitalIO.ioDevice('XI-1022', ioVar22);
    let ioData23 = new digitalIO.ioDevice('XI-1023', ioVar23);
    let ioData24 = new digitalIO.ioDevice('XI-1024', ioVar24);
    let ioData25 = new digitalIO.ioDevice('XI-1025', ioVar25);
    let ioData26 = new digitalIO.ioDevice('XI-1026', ioVar26);
    let ioData27 = new digitalIO.ioDevice('XI-1027', ioVar27);
    let ioData28 = new digitalIO.ioDevice('XI-1028', ioVar28);
    let ioData29 = new digitalIO.ioDevice('XI-1029', ioVar29);
    let ioData30 = new digitalIO.ioDevice('XI-1030', ioVar30);
    let ioData31 = new digitalIO.ioDevice('XI-1031', ioVar31);
    let ioData32 = new digitalIO.ioDevice('XI-1032', ioVar32);
    // Creates analog device (object) using class constructor
    aiDevice5 = new analogIO.aiDevice('PIT-650', ai5Voltage, 0, 6000, 'pa', 5000, 0, 0, 0,
4000, 0, 0, 0, 5, 0);
    aiDevice6 = new analogIO.aiDevice('PIT-730', ai6Voltage, 0, 100, 'kpa', 0, 0, 0, 0, 0,
0, 0, 0, 5, 0);
    aiDevice7 = new analogIO.aiDevice('TIT-990', ai7Voltage, 0, 200, 'deg C', 0, 0, 0, 0,
0, 0, 0, 0, 5, 0);
    aiDevice8 = new analogIO.aiDevice('FIT-828', ai8Voltage, 0, 150, 'deg C', 0, 0, 0, 0,
0, 0, 0, 0, 5, 0);
    // Returns Scaled PV for sending to Database in above scan, scanAnalog not required
here therefore

    // Create a new data map of the discrete and analog IO TAG and Values for database
    var ioDataMap = new Map();
    ioDataMap.set(ioData17.tag, ioData17.raw);
    ioDataMap.set(ioData18.tag, ioData18.raw);
    ioDataMap.set(ioData19.tag, ioData19.raw);
    ioDataMap.set(ioData20.tag, ioData20.raw);
    ioDataMap.set(ioData21.tag, ioData21.raw);
    ioDataMap.set(ioData22.tag, ioData22.raw);
    ioDataMap.set(ioData23.tag, ioData23.raw);
    ioDataMap.set(ioData24.tag, ioData24.raw);
    ioDataMap.set(ioData25.tag, ioData25.raw);
    ioDataMap.set(ioData26.tag, ioData26.raw);
    ioDataMap.set(ioData27.tag, ioData27.raw);
    ioDataMap.set(ioData28.tag, ioData28.raw);
    ioDataMap.set(ioData29.tag, ioData29.raw);
    ioDataMap.set(ioData30.tag, ioData30.raw);
    ioDataMap.set(ioData31.tag, ioData31.raw);
    ioDataMap.set(ioData32.tag, ioData32.raw);
    ioDataMap.set(aiDevice5.tag, aiDevice5.scaledPV.toFixed(1));
    ioDataMap.set(aiDevice6.tag, aiDevice6.scaledPV.toFixed(1));
    ioDataMap.set(aiDevice7.tag, aiDevice7.scaledPV.toFixed(1));
    ioDataMap.set(aiDevice8.tag, aiDevice8.scaledPV.toFixed(1));
    //console.log('ioData values are:', ioDataMap);

    // Section for checking alarm activation of device - Only 1 Analog Device being tested
here currently
    var aiDev1param = aiDevice5.checkAlarms();
    // Below console output used for testing and visibility only
    console.log('Scaled PV is: ', aiDev1param[0].toFixed(0), ' | Alarm Flag is: ',
aiDev1param[1], ' | Alarm Status is: ', aiDev1param[2], ' | Device Tag is: ',
aiDev1param[3]);

    // Create Alarm data if alarmFlag generated in above alarmCheck() method call for each
device
    var alarmDataMap = new Map();
    if (aiDev1param[1] == 'true' && alarmFlag == false) {
        alarmDataMap.set(aiDevice1.tag, aiDev1param[2]);
        alarmFlag = true;
        transmitFlag = true;
    }
    else if (aiDev1param[1] == 'false' && alarmFlag == true) {
        // If aiDevice1.resetAlarm() method is called the aiDev1param[1] is set false and
this will run resetting the logging to DB
        alarmFlag = false;
    }
    else {
        // creates transmit one-shot
        transmitFlag = false;
        // Logs nothing to Database until the alarm is reset
    }

    // Converts the created data map of analog and discrete tag/values to JSON for
websocket transmit
    const obj1 = Object.fromEntries(ioDataMap);
    const myioData = JSON.stringify(obj1);
    // Converts the created data map of alarm tag/values to JSON for websocket transmit
    const obj2 = Object.fromEntries(alarmDataMap);
    const myAlarmData = JSON.stringify(obj2);
    const wsflag = JSON.stringify(transmitFlag);

    return([myioData, myAlarmData, wsflag]);
};

module.exports = {
    updateIO, // used within client/1/2.js

};
```

D.4.12 devices/clientAnalog.js code

```
// Client Interface function and class constructors for Analog data IO elements

// Global flags used for promise passing of returned values within sections of overall
object methods
var HHflag = 'false';
//var Hflag = 'false';
//var Lflag = 'false';
//var LLflag = 'false';
// Main Analog Interface object - allows object assignment via constructor, provides
internal and external methods for readings, alarms etc
class aiDevice {

    /* Class Constructor - Used for creating analog input instances
       Inputs are as stated with High High, High, Low, Low Low parameters and delay (d_HH
etc) in mS for each.
       Any alarms paramaters 'HH, H, L, LL' assigned 0 are disabled. */
    constructor(tag, raw, eng_lrv, eng_urv, units, HH, H, L, LL, d_HH, d_H, d_L, d_LL,
maxRaw, minRaw) {
    this.tag = tag;
    this.raw = raw;
    this.eng_lrv = eng_lrv;
    this.eng_urv = eng_urv;
    this.units = units;
    this.HH = HH;
    this.H = H;
    this.L = LL;
    this.LL = LL;
    this.d_HH = d_HH;
    this.d_H = d_H;
    this.d_L = d_L;
    this.d_LL = d_LL;
    this.maxRaw = maxRaw;                   // 5 volt input signal maximum
    this.minRaw = minRaw;                   // 0 volt input signal minimum
    this.HHtimeoutID = 'HH_Alarm_Timeout';
    this.HtimeoutID = 'H_Alarm_Timeout';
    this.LtimeoutID = 'L_Alarm_Timeout';
    this.LLtimeoutID = 'LL_Alarm_Timeout';
    this.alarmFlag = 'false';               // Active if alarm generated
    this.alarmTimeoutFlag = 'false';        // Ensures alarm timeout only activated once
    this.scaledPV = (this.eng_urv-this.eng_lrv)*((this.raw-this.minRaw)/this.maxRaw);
    }

    // Getter - Returns current PV in engineering units
    get currentPV() {
        this.scaledPV = (this.eng_urv-this.eng_lrv)*((this.raw-this.minRaw)/this.maxRaw);
        console.log('Output for ' + this.tag + ' is: ' + this.scaledPV.toFixed(0) + ' ' +
this.units)
    }
    // Method - Update raw process variable input and return scaled Process Variable
    updateRaw(raw) {
        this.raw = raw;
        this.scaledPV = (this.eng_urv-this.eng_lrv)*((this.raw-this.minRaw)/this.maxRaw);
        console.log('raw is: ' + this.raw.toFixed(2) + ' volts DC.', 'Scaled PV is: ' +
this.scaledPV.toFixed(0) + ' kpa');
    }

    // Method - Used to reset alarms on device after activation
    resetAlarm(reset) {
        this.alarmFlag = reset; // value = false passed into object to reset flag
    }

    // ------- Async Operations to check delay timers and return promise when done ------//

    // Inner Method - Used for creating Timeout with Promise for Method checkAlarm()
    delayTimer(t, timeoutID) {
        return new Promise(function(resolve) {
            timeoutID = setTimeout(function() {
                resolve();
            }, t);
        });
    }

    // Inner Method - Encapsulates the promise and allows returns 'true' which is assigned
in below execution invocation
    setHHAlarmFlag() {
        return this.delayTimer(this.d_HH,'HH_Alarm_Timeout').then(function() {
            return 'true';
        });
    }

    // Set further H, L, LL alarm flag methods here...

    // ---------------------------------------------------------------------------------
//
    // Method - checks the current PV against all assigned alarms and delays
    checkAlarms() {
        this.alarmData;
        // ------------------ HH alarm checks ------------------ //
        if(this.HH != 0) {
            console.log('Checking HH alarms status, ',' ScaledPV is: ',
this.scaledPV.toFixed(0),'HH Setpoint is: ', this.HH, 'Alarm Flag is: ', this.alarmFlag);
            //console.log('ScaledPV is: ', this.scaledPV.toFixed(0),'HH Setpoint is: ',
this.HH, 'Alarm Flag is: ', this.alarmFlag);

            // ------------------ HH Alarm with Delay checks ------------------ //
            if (this.d_HH != 0 && this.scaledPV > this.HH) {

                // One-shot delay timer activation for HH with delay
                if(this.alarmTimeoutFlag == 'false') {

                    // Ensures timer only activated once until reset by healthy PV
                    this.alarmTimeoutFlag = 'true';
                    console.log('******** HH alarm active with delay, timeout activated
********');
                    // Promise used to ensure timer has expired before writing to
'this.alarmFlag'
                    // Execution - This runs the function and gives access to the returned
inputted data upon promise resolution
                    this.setHHAlarmFlag(HHflag).then(function(returnedValue) {
                        console.log('**** Promise value returned is: ', returnedValue);
                        HHflag = 'true';
                    })
                    console.log('**** alarmflag value in "if" call: ', this.alarmFlag);
                }
                // Runs after one-shot code above has activated the async delay timeout
                else {
                    console.log('**** HHflag value is: ', HHflag);
                    console.log('**** Alarmflag value in "else" call: ',
this.alarmFlag);
                }

                // If Promise flag returns true then set alarmFlag true to activate alarm
                if (HHflag == 'true') {
                    this.alarmFlag = 'true';
                    this.alarmData = ('High High Alarm @ ' + this.HH + this.units + ' with
' + this.d_HH + 'mS delay');
                    console.log('** TIMEOUT HAS OCCURED - HIGH HIGH ALARM ACTIVATED **')
                }
                else {
                    this.alarmFlag = 'false';
                    console.log('Alarm Promise timeout incrementing');
                }
                console.log('Alarm flag post HHflag check value is: ', this.alarmFlag);

            }

            // ------------------ Instant HH Alarm with No Delay checks ------------------
//
            else if(this.d_HH == 0 && this.scaledPV > this.HH) {
                console.log('INSTANT HIGH HIGH ALARM');
                this.alarmFlag = 'true';
                this.alarmData = ('High High Alarm @ ' + this.HH + this.units);
            }
            // ------------------ HH Alarm with Delay Healthy SP checks ------------------
//
            else if(this.d_HH != 0 && this.scaledPV < this.HH) {
                console.log('HH Alarm clear');
                this.alarmData = 'Healthy';
                this.alarmTimeoutFlag = 'false';    // Allows the alarm delay timeout to be
activated again if PV in alarm range again
                clearTimeout('HH_Alarm_Timeout');   // Clears the active delay timeout as
PV is healthy again
            }
            // ----------------------- HH Alarm Not Enabled checks -----------------------
//
            else {
                console.log('HH Alarm not enabled');
                //this.alarmFlag = 'false';
                this.alarmData = 'Disabled';
            }
        }
        // ****** Add remaining H alarm checks and further L & LL alarms here ****** //

    // End of checkAlarm() Method returns the current scaled PV and alarm status for user
    return([this.scaledPV, this.alarmFlag, this.alarmData, this.tag]);
    };

};

// Function definition which calls object methods to update input to analog object,
// returns the scaled PV and assesses and activates alarms
function scanAnalog(aiDevice) { setInterval(() => {
    aiDevice.currentPV;
    aiDevice.updateRaw(aiDevice.raw);
    var par = aiDevice.checkAlarms();
    console.log('Scaled PV is: ', par[0].toFixed(0), ' | Alarm Flag is: ', par[1], ' |
Alarm Status is: ', par[2], ' | Device Tag is: ', par[3]);
    }, 2000)
};

// Scans the analog device as per above function at set interval time period below example
invocation
//scanAnalog(aiDevice1);
module.exports = {
    aiDevice,
    scanAnalog
};
```

D.4.13 devices/clientDigital.js code

```
// Client Interface function and object constructors for Discrete data IO
elements

// Simple object definition for generic Discrete IO device
function ioDevice(tag, raw) {
    this.tag = tag;
    this.raw = raw;
}

function LL_Alarm(tag, raw) {
    this.tag = tag;
    this.raw = raw;
}

module.exports = {
    ioDevice,
};
```

D.4.14 devices/clientSendData.js code

```
// Functions here are used to take individual client data and send over their websocket
connection

// Import time functions from the date.js module
const currentTime = require('../date.js');

/* The below clientSendData takes the client ID, socket connection and passes a function
inside
   which runs. This is where the individual client data READ and UPDATE operations are
called internally */

function clientSendData (clientID, socket, sendDataFunction){
    // Below is where the individual client updateIO() functions which are passed in are
called
    var ioData = sendDataFunction();
    let dateTimeGMT = currentTime.timeStamp();
    dateTime = dateTimeGMT[0].toString();
    // The data read and returned by the above function is then sent using WS to Server
    if (socket.bufferedAmount == 0) {

        socket.send(JSON.stringify({type: 'io', clientID: clientID, time: dateTime, data:
ioData[0]}), function () {
        });
        // If one-shot alarm flag is set then transmit the alarm data across WS to the DB
        if (ioData[2] == 'true') {
            socket.send(JSON.stringify({type: 'alarm', clientID: clientID, time: dateTime,
data: ioData[1]}), function () {
            });
        }
        else {
            // Do nothing - no transmit of alarms needed to database, already logged
        }
    } else {
        console.log('Websockets buffer full');
    }
};

// The below generic function calls the above at the set interval (This is exported to
clientSocket.js)
function sendData(clientID, socket, sendDataFunction) { setInterval(() =>
clientSendData(clientID, socket, sendDataFunction), 500)};

module.exports = {
    sendData
}
```

D.4.15 devices/serverFormatData.js code

```
// Function for formatting JSON data prior to writing to database
function formatDBdata (ioData) {
    //console.log('iodata input as JSON is:',ioData)
    let parsedIOdata = JSON.parse(ioData)
    var keyArray = [];
    var valueArray = [];
    Object.keys(parsedIOdata).forEach(key => {
        //console.log('key', key);
        keyArray.push(JSON.stringify(key));
    });
    Object.values(parsedIOdata).forEach(value => {
        //console.log('value', value);
        valueArray.push(JSON.stringify(value));
    });
    //console.log(typeof keyArray,keyArray);
    //console.log(valueArray);
    return{keyArray, valueArray};
};

module.exports = {
    formatDBdata
}
```

D.4.16 controllers/error.js code

```
// Below is used to direct to the 404 HTML page if an error is encountered during
navigation

exports.get404 = (req, res, next) => {
    res.status(404).render('hmi/404', {
   pageTitle: 'Page Not Found',
   path: '/404'
   });
};
```

D.4.17 controllers/hmi.js code

```
// Below is used to direct to the main HMI landing page for the application

const devices = require('../models/devices'); // imports the class from the models folder

var Device1_PV;
Device1 = new devices.Device('PIT-210', 'Pressure Transmitter');

/* Allow time for databse to read first client data then call Process Variable
   using class method for the model to send to webpage as displayed value */
setTimeout(() => {
    Device1_PV =  Device1.fetchPV;
  }, 15000);

// Render requested page as directed through router with model included
exports.getIndex = (req, res, next) => {
            res.render('hmi/index',{
                devs: Device1,
                pageTitle: 'HMI Landing Page',
                path: '/'
            });
    };
```

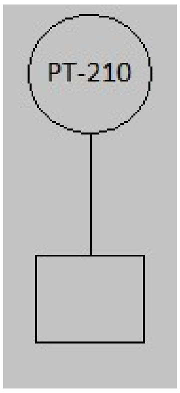
D.4.18 models/devices.js code

```
// The below describes a new model which is then exported and used for database storage

// Import io write to DB function from database.js module
const { json } = require('body-parser');
var sqlDB = require('../database.js');

/* Class for models used as part of the MVC structure. This has an internal method
'HMI_Data' which
   is called within a getter function allowing the Process Variable to be rendered on HTML
page */
class Device {
    constructor(tag, description) {
        this.tag = tag;
        this.description = description;
        this.PV;
    }

    // Method - Call database current PV for Device
    HMI_Data() {
        this.PV = sqlDB.ioDataFromDB();
        //console.log('****** val value is *******', this.PV);
        //return this.PV;
    }

    // Getter - Calls above HMI_Data
    get fetchPV() { setInterval(() => this.HMI_Data(), 1500)
        return this.PV;
     }
}

module.exports = {
    Device
};
```

D.4.19 public/css/main.css code

```
body {
    background-color: lightblue;
  }

  div {
    background-color: lightblue;
  }
```

D.4.20 public/images/mockPT.png



D.4.21 routes/hmi.js code

```
// Routing page for the main HMI landing / test page

// Import dependencies
const path = require('path');
const express = require('express');

const hmiController = require('../controllers/hmi.js');
const router = express.Router();

// The below is called during runtime of the main 'app.js' application
// This initially calls the controller listed below which directs to the index.ejs page
// localhost:3000 => GET (routes to the EJS file specified and passes any listed variables
as listed)
router.get('/', hmiController.getIndex);

module.exports = router;
```

D.4.22 views/hmi/404.ejs code

```
<%- include('../includes/head.ejs') %>

<body>
    <h1>This is an error 404 page</h1>

<%- include('../includes/end.ejs') %>
```

D.4.23 views/hmi/index.ejs code

```
<%- include('../includes/head.ejs') %>

<body class="body">
    <main>
        <h1 class="header_1">Main HMI Test / Landing Page</h1>
        <div>
        <article>
            <header class="header_1">
                <h1 class="title"><%= devs.tag %></h1>
            </header>
            <div>
                <img class="image" src="/images/mockPT.png" alt="<%= devs.tag %>">
            </div>
            <div>
                <h2 class="header_2"><%= devs.PV %></h2>
                <p class="description"><%= devs.description %></p>
            </div>
        </article>
        </div>
        </main>



<%- include('../includes/end.ejs') %>
```

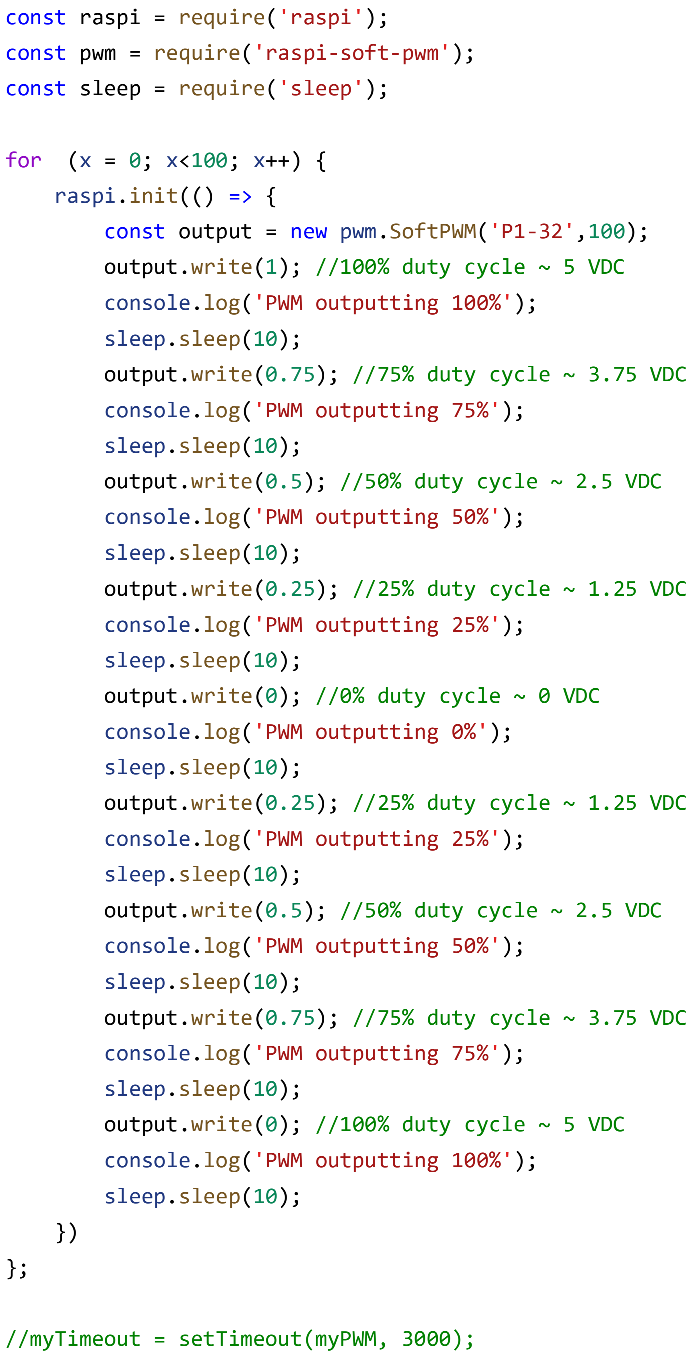
D.4.24 views/includes/head.ejs code

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <!-- The below line refreshes the data from model every second
    this is not optimal method but allows confirmation of operations -->
    <meta http-equiv="refresh" content="1">

    <meta name="viewport" content="width=device-width, intial-scale=1.0">
    <title><%= pageTitle %></title>
    <link rel="stylesheet" href="/css/main.css">
</head>
```

D.4.25 views/includes/end.ejs code

```
<script src="/js/main.js"></script>

</body>

</html>
```

D.4.26 pwm.js code (Tested as individual Node executable)



```
const raspi = require('raspi');
const pwm = require('raspi-soft-pwm');
const sleep = require('sleep');

for  (x = 0; x<100; x++) {
    raspi.init(() => {
        const output = new pwm.SoftPWM('P1-32',100);
        output.write(1); //100% duty cycle ~ 5 VDC
        console.log('PWM outputting 100%');
        sleep.sleep(10);
        output.write(0.75); //75% duty cycle ~ 3.75 VDC
        console.log('PWM outputting 75%');
        sleep.sleep(10);
        output.write(0.5); //50% duty cycle ~ 2.5 VDC
        console.log('PWM outputting 50%');
        sleep.sleep(10);
        output.write(0.25); //25% duty cycle ~ 1.25 VDC
        console.log('PWM outputting 25%');
        sleep.sleep(10);
        output.write(0); //0% duty cycle ~ 0 VDC
        console.log('PWM outputting 0%');
        sleep.sleep(10);
        output.write(0.25); //25% duty cycle ~ 1.25 VDC
        console.log('PWM outputting 25%');
        sleep.sleep(10);
        output.write(0.5); //50% duty cycle ~ 2.5 VDC
        console.log('PWM outputting 50%');
        sleep.sleep(10);
        output.write(0.75); //75% duty cycle ~ 3.75 VDC
        console.log('PWM outputting 75%');
        sleep.sleep(10);
        output.write(0); //100% duty cycle ~ 5 VDC
        console.log('PWM outputting 100%');
        sleep.sleep(10);
    })
};

//myTimeout = setTimeout(myPWM, 3000);
```

D.4.27 adcpi.js code (Tested individually and within updateIO() function)

```javascript
var adcpi = require('./ABElectronics_NodeJS_Libraries/lib/adcpi/adcpi');

var adc = new ADCPi(0x68, 0x69, 18);

setInterval (function() {

    let value = adc.readVoltage(1);
    let raw = adc.readRaw(1);

    console.log(value);
    console.log(raw);
},2000);
```