University of Southern Queensland

Faculty of Health, Engineering and Sciences

Low-cost compact Modbus TCP based device configuration/interrogation tool

A dissertation submitted by

Ethan Picking

in fulfilment of the requirements of

ENG4111 and 4112 Research Project

towards the degree of

Bachelor of Engineering (Electrical and Electronic)

Submitted October, 2020

Abstract

Modbus TCP is a common communications protocol used by many industrial devices and control systems. Many modern devices that include Modbus TCP capability contain a significant number of configuration parameters and feedback data points that may be accessed using this capability. Often these devices are installed as replacements for much simpler redundant equipment, in locations that lack the control systems or network infrastructure to utilise the Modbus TCP capability. Additionally, the large number of configuration parameters used by these newer devices makes them much more difficult and time consuming to configure/commission than the simpler devices they replace.

This dissertation documents the development and design of a Configuration and Interrogation tool that operates over the Modbus TCP protocol. The final hardware design of this tool is based upon the Raspberry PI Model 3B+ platform, and utilises software modules developed in the Python 3 programming language. The final outcome of this project is an operational prototype that has been tested against a selected Modbus TCP device and meets the following primary objectives. This final prototype meets the primary objectives of this project in its ability to perform the required configuration and interrogation functions, while remaining portable and low-cost.

University of Southern Queensland

Faculty of Health, Engineering and Sciences

ENG4111/ENG4112 Research Project

Certification of Dissertation

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

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

E. Picking

[redacted]

Acknowledgements

I would like to thank all the USQ staff that have given freely of their knowledge and assistance over the course of my studies.

Specifically, thanks to my supervisor for this project, Mr Mark Phythian, for his guidance and other assistance throughout its execution.

Lastly, thanks to Ron and Julie, for supporting me along this path.

Contents

Abstract i

Acknowledgements iv

List of figures ix

List of tables xi

Nomenclature xii

1 Introduction 1

1.1 Project Objectives 2

1.2 Project Structure 3

2 Literature Review 4

2.1 Modbus Background 4

2.2 Modbus Application Protocol Overview 6

2.3 Modbus TCP/IP Ethernet Specifics 9

2.4 Modbus Error Handling 11

2.5 Existing Modbus Communications Options 12

2.6 SEL-710 Motor Protection Relay 13

3 Analysis of Hardware and Software Requirements 15

3.1 Consideration of Consequential Effects 15

3.2 SEL-710 Hardware Considerations 16

3.3 Hardware Requirements 17

3.4 Available Hardware 18

3.4.1 Arduino Uno 19

3.4.2 Raspberry Pi 3B+ 20

3.4.3 BeagleBone Black 21

3.5 Hardware Platform Selection 22

3.6 SEL-710 Software Considerations 23

3.7 Required Modbus Function Codes 23

3.8 Software Requirements 24

4 Hardware Design 27

4.1 SEL-710 Test Unit Design 27

4.2 Configuration Tool Functional Design 28

4.3 Raspberry Pi 3B+ 29

4.4 HDMI Touch Display 30

4.5 Battery Power Supply 31

4.6 Enclosure 32

4.7 Bill of Materials 32
5 Software Design 34
5.1 Overall Structure 34
5.2 Device Configuration File Format 38
5.3 Parameter Settings File Format 39
5.4 Interrogation Log File Format 40
5.5 File Input/Output 40
5.5.1 Import Device Configuration (Read_Config) 40
5.5.2 Import Parameter Settings (Read_Settings) 41
5.5.3 Export Log File (Write_Log) 41
5.6 Modbus Address Encoding/Decoding 41
5.6.1 Addr_Settings 41
5.6.2 Addr_Lookup 42
5.7 Modbus Format 42
5.7.1 __init__ 43
5.7.2 __repr__ 43
5.7.3 __bytes__ 43
5.7.4 set_PKT 43
5.7.5 set_TX_ID 43
5.7.6 set_PRTCL 44
5.7.7 set_UN_ID 44
5.7.8 set_FN_CD 44
5.7.9 set_DATA 44
5.8 TCP Communications 45
5.8.1 Comm 45
5.8.2 Gen_Requests 45
5.9 Modbus Error Handling 46
5.9.1 Modbus_Error 46
5.10 Modbus Read request/response handling 48
5.10.1 Gen_Read_Discrete_Output_Coil_PDUS (Function Code 0h01) 48
5.10.2 Gen_Read_Discrete_Output_Coil_Reply (Function Code 0h01) 49
5.10.3 Gen_Read_Discrete_Input_Status_PDUS (Function Code 0h02) 50
5.10.4 Gen_Read_Discrete_Input_Status_Reply (Function Code 0h02) 50
5.10.5 Gen_Read_Holding_Registers_PDUS (Function Code 0h03) 50
5.10.6 Gen_Read_Holding_Registers_Reply (Function Code 0h03) 51
5.10.7 Gen_Read_Input_Registers_PDUS (Function Code 0h04) 52

5.10.8 Gen_Read_Input_Registers_Reply (Function Code 0h04) 52
5.11 Modbus Write request/response handling 53
5.11.1 Gen_Single_Coil_Force_PDU (Function Code 0h05) 53
5.11.2 Gen_Single_Coil_Force_Reply (Function Code 0h05) 54
5.11.3 Gen_Preset_Single_Register_PDU (Function Code 0h06) 54
5.11.4 Gen_Preset_Single_Register_Reply (Function Code 0h06) 55
5.11.5 Gen_Preset_Multiple_Registers_PDUS (Function Code 0h10) 55
5.11.6 Gen_Preset_Multiple_Registers_Reply (Function Code 0h10) 56
5.12 Device configuration 57
5.12.1 Modbus_Configure 57
5.12.2 User_Config_Input 58
5.13 Device Interrogation 58
5.13.1 Modbus_Interrogate 59
5.13.2 User_Interrogate_Input 59
5.14 Raspberry Pi Initial Network Configuration 60
6 Testing 61
6.1 Configuration Tests 62
6.1.1 Phase One 62
6.1.2 Phase Two 63
6.2 Interrogation Tests 64
6.3 Battery Tests 66
7 Conclusions and Further Work 68
7.1 Further Work 69
8 References 70
Appendix A, Project Specification 72
Appendix B, Device Code 73
B.1 File Input/Output Module Code 73
B.2 Modbus Address Encoding/Decoding Module Code 75
B.3 Modbus Format Module Code 78
B.4 TCP Communications Module Code 83
B.5 Modbus Error Handling 86
B.6 Modbus Read request/response handling Module Code 87
B.7 Modbus Write request/response handling 100
B.8 Device configuration 109
B.9 Device Interrogation 112
Appendix C, Risk Assessment 119

Appendix D, Example Typical Use Parameter Set 120

Appendix E, Example Log File Excerpt 121

Appendix F, User Instructions 122

List of figures

Figure 2.1 Modbus communication stack (Modbus Organisation 2012) 4
Figure 2.2 Example of MODBUS Network Architecture (Modbus Organisation 2012) 5
Figure 2.3 Standard Successful Modbus Transaction (Modbus Organisation 2012).......... 6
Figure 2.4 Standard Unsuccessful Modbus Transaction (Modbus Organisation 2012) 7
Figure 2.5 General Modbus Frame (Modbus Organisation 2012) 7
Figure 2.6 Standard Modbus Function Codes (Modbus Organisation 2012).......... 8
Figure 2.7 Example Modbus TCP/IP Network Layout (Modbus Organisation 2006) 9
Figure 2.8 Modbus TCP/IP request/response format (Modbus Organisation 2006) 10
Figure 2.9 Standard Modbus transaction (Modbus Organisation 2012) 11
Figure 2.10 The SEL-710 Motor Protection Relay (Schweitzer Engineering Laboratories 2015) 13
Figure 3.1 SEL-710 ELV Model power supply requirements (Schweitzer Engineering Laboratories 2015) 16
Figure 3.2 SEL-710 Terminal arrangement and assignment (Schweitzer Engineering Laboratories 2015) 17
Figure 3.3 Arduino Uno (Arduino AG 2020).......... 19
Figure 3.4 Raspberry Pi 3B+ (Raspberry Pi Foundation 2020) 20
Figure 3.5 BeagleBone Black (BeagleBoard.org Foundation 2019).......... 21
Figure 3.6 SEL-710 Accepted Modbus Function Codes (Schweitzer Engineering Laboratories 2015) . 23
Figure 3.7 Modbus Client Activity Diagram (Modbus Organisation 2006) 25
Figure 4.1 SEL-710 Test unit schematic 27
Figure 4.2 Modbus Configuration Device Schematic 28
Figure 4.3 Assembled Prototype Hardware.......... 29
Figure 4.4 Raspberry Pi Model 3B 29
Figure 4.5 Waveshare 5-Inch Touchscreen Display.......... 30
Figure 4.6 Battery Power Supply Module 31
Figure 4.7 Takachi MXA Series Enclosure (Takachi Electronics Enclosure Co. Ltd. 2020) 32
Figure 5.1 Configuration Function Flowchart 36
Figure 5.2 Interrogation Function Flowchart 37
Figure 5.3 Standard Configuration File Entry.......... 38
Figure 5.4 Bit Descriptor Configuration File Entry.......... 39
Figure 5.5 Settings File Entries 39
Figure 5.6 Log File Entries 40
Figure 5.7 Modbus Function Code 0h01 Request Format (Modbus Organisation 2012).......... 49

Figure 5.8 Modbus Function Code 0h01 Response Format (Modbus Organisation 2012) 49
Figure 5.9 Modbus Error Response Format (Modbus Organisation 2012).. 49
Figure 5.10 Modbus Function Code 0h02 Request Format (Modbus Organisation 2012).................. 50
Figure 5.11 Modbus Function Code 0h02 Response Format (Modbus Organisation 2012) 50
Figure 5.12 Modbus Function Code 0h03 Request Format (Modbus Organisation 2012).................. 51
Figure 5.13 Modbus Function Code 0h03 Response Format (Modbus Organisation 2012) 52
Figure 5.14 Modbus Function Code 0h04 Request Format (Modbus Organisation 2012).................. 52
Figure 5.15 Modbus Function Code 0h04 Response Format (Modbus Organisation 2012) 53
Figure 5.16 Modbus Function Code 0h05 Request Format (Modbus Organisation 2012).................. 54
Figure 5.17 Modbus Function Code 0h05 Response Format (Modbus Organisation 2012) 54
Figure 5.18 Modbus Function Code 0h06 Request Format (Modbus Organisation 2012).................. 55
Figure 5.19 Modbus Function Code 0h06 Response Format (Modbus Organisation 2012) 55
Figure 5.20 Modbus Function Code 0h10 Request Format (Modbus Organisation 2012).................. 56
Figure 5.21 Modbus Function Code 0h10 Response Format (Modbus Organisation 2012) 57
Figure 5.22 Raspberry Pi Network Configuration.. 60
Figure 6.1 Testing Hardware Connections 61
Figure 6.2 Prototype tool under test 62
Figure 6.3 Prototype tool during Interrogation testing 66

List of tables

Table 3.1 Requirement Priorities 18

Table 3.2 Comparison of Hardware Platforms 22

Table 5.1 Configuration File Entry Fields 38

Table 5.2 Modbus Exception Codes (Modbus Organisation 2012) 47

Table 6.1 Configuration Routine Phase One Testing Results 63

Table 6.2 Manual Configuration vs Tool Configuration Routine Phase Two Testing Results 64

Table 6.3 Interrogation Routine Test Results 65

Table 6.4 Battery Life Test Results 67

Table C.1 Consequence vs. Likelihood Risk Matrix........ 119

Table C.2 Project Risk Assessment........ 119

Table D.1 Typical Use Parameter Set........ 120

Table E.1 Interrogation Routine Log File Excerpt........ 121

Nomenclature

ADU	Application Data Unit
CSV	Comma Separated-Values File Format
IP	Internet Protocol
MBAP	Modbus Application Protocol
MPR	Motor Protection Relay
PDU	Protocol Data Unit
SEL	Schweitzer Engineering Laboratories
TCP	Transmission Control Protocol
YAML	YAML Ain’t Markup Language File Format

1 Introduction

The purpose of this project is to develop and document a low-cost Modbus TCP device configuration and interrogation tool. It is intended that this device will be compact, and constructed from a readily available hardware platform such as a Raspberry Pi, or Arduino, and operate as a self-contained portable unit with integral power supply and user interface. The purpose of this device will be to interface with Modbus TCP capable equipment that is installed in locations where a Modbus control network is not present. This in turn will enable the inbuilt Modbus TCP functionality to be leveraged for ease of configuration, commissioning, parameter verification, and fault diagnosis. By enabling use of these functions via a simplified user interface, preferably graphical in nature, that provides sufficient guidance to allow use by an individual that is not familiar with the specifics, operation, or technical detail of the Modbus protocol. Specifically, this device will be targeted for use with protection relays, but allowance will be made to allow its configuration for use with other Modbus TCP equipment, using standard Modbus functions.

The origin of this project is sourced in the author's previous experience installing, commissioning, and diagnosing modern motor protection relays used as replacements for older electro-mechanical or early model electronic relays. In these situations, the modern relays almost universally support Modbus TCP, but due to the lack of control infrastructure capable of utilising this capability it often goes underutilised. Thus, for the purposes of this project a Modbus TCP capable motor protection relay will be used as the test case, specifically a Schweitzer Engineering Laboratories Model SEL-710.

The wider range of protection features in modern protection relays has resulted in large configuration parameter sets with hundreds of elements (Schweitzer Engineering Laboratories 2015). The most efficient method available to set these parameters is often via a laptop computer and vendor specific software. This provides the required functions, but the software is generally targeted at engineering level personnel, and does not provide a simple interface option that would be accessible to personnel without more advanced training or experience. This necessitates that these tasks be performed generally by technical engineering personnel, who may not be readily available, for example in the instance of a fault occurring outside of regular working hours. The use of a laptop computer to perform this function can also prove sub optimal, by virtue of their weight and size, as well as their cost and sensitivity to physical shock and damage in an industrial setting.

Relays may also offer a manual user interface option to configure parameters without use of external equipment, which is performed via a local display and selection of push buttons, thought this method may not allow access to the full parameter set. This method is often more accessible to maintenance personnel but due to the limitations of the interface, which may only consist of basic scrolling character display, and directional arrow/enter keys. As a result, this method can become extremely time consuming and error prone due to inadvertent key strokes, which in turn necessitates further time be spent to confirm the correct settings have been entered. This method also provides minimal to no assistance with fault diagnosis, consisting perhaps of only a list of recent trip causes and perhaps some accompanying snapshot data of key measurements.

It follows that a compact low-cost device capable of performing configuration and diagnostic functions, whilst remaining accessible to electrical maintenance personnel without higher engineering training, would be of significant advantage in a wide range of situations.

1.1 Project Objectives

Specifically, the developed tool must satisfy the following objectives in order to be considered successful:

- Compact and portable in nature
- Capable of direct hardware connection to a Modbus TCP device, necessitating an RJ-45 port be included
- Hardware to be low cost
- Hardware to be readily available
- Have the capability to configure a Modbus TCP device, given a user supplied settings/parameter file
- Have the capability to interrogate/monitor specific Modbus TCP device data, given a user supplied query
- Capable of being configured for use with a wide range of Modbus TCP devices

1.2 Project Structure

In order to achieve these objectives, the project has been ordered into the following three phases:

Phase 1. Research and Review

- Review available literature Modbus TCP specifications, principle of operation, and standard methods of implementation.
- Obtain and review relevant documentation specific to the chosen Modbus TCP capable device; a Schweitzer Engineering Laboratories model SEL-710 motor protection relay.

Phase 2. Conceptual Design and Hardware Selection

- Determine standard functions that would be required to be implemented over Modbus TCP for the purpose of programming, commissioning, and fault diagnosis.
- Obtain the selected SEL-710 MPR test unit and assemble with a suitable enclosure and power supply into a suitable test platform.
- Determine the hardware and software requirements of a device to interface with the selected Modbus TCP device.
- Select, obtain, and assemble suitable hardware that meets the determined hardware requirements, is resilient to physical damage, and integrates a portable power supply with provision for a simple method of recharge.

Phase 3 Software Development and Testing

- Develop an effective software platform capable of performing the required Modbus TCP functionality to suit the assembled hardware platform for the performance of programming, commissioning, and interrogation operations.
- Perform functional testing of the developed device utilising the test device assembly.

2 Literature Review

This chapter provides a review of the relevant available literature for the following concepts:

1. Modbus Background
2. Modbus Application Protocol Overview
3. Modbus TCP/IP Ethernet Specifics
4. Modbus Error Handling
5. Existing Modbus Communication Options
6. SEL-710 Motor Protection Relay

2.1 Modbus Background

Modbus was originally developed by Gould Modicon as a transmission protocol for process control systems (Park et al. 2004). Reynders et al. (2004) describes Modbus as 'an application layer (OSI layer 7) messaging protocol that provides client/server communication between devices'. Despite being released in 1979 the protocol has remained popular across multiple device types and vendors, due in large part to the fact that it is publicly published, providing a protocol that vendors can easily standardise too. As the protocol is not restricted to a specific physical interface it has been adapted to several, providing great flexibility, ranging from EIA-232 Serial to TCP (Transmission Control Protocol) over Ethernet (Reynders et al. 2004).

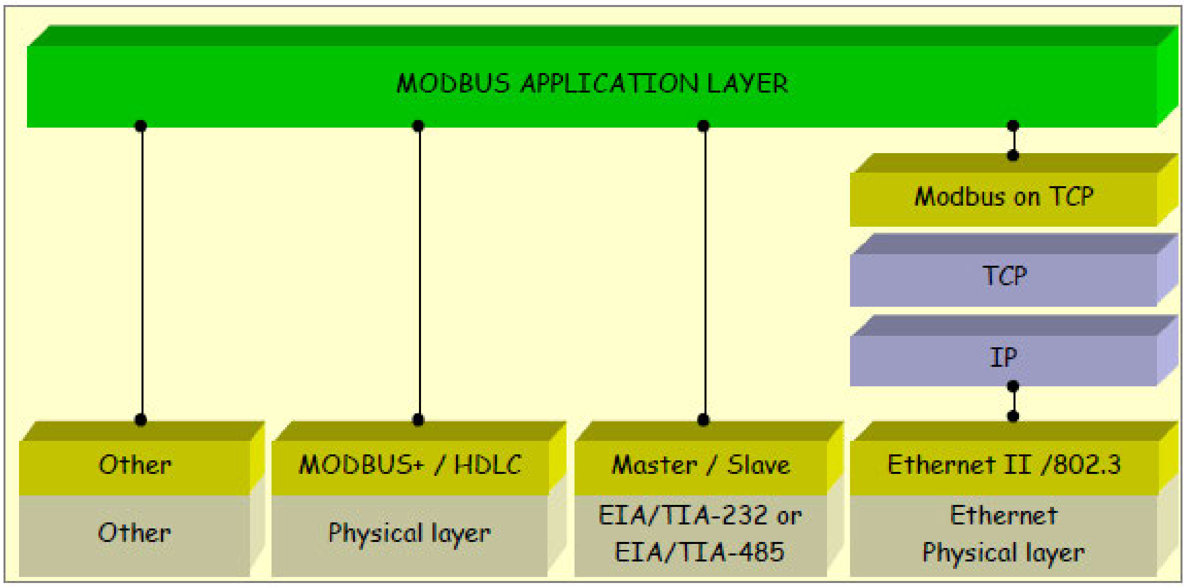


Figure 2.1 Modbus communication stack (Modbus Organisation 2012)

This flexibility, as well as the simplicity of maintaining the same protocol across a range of physical interfaces, has ensured the ongoing success of the Modbus protocol. Essentially customers seek Modbus capable devices to ensure compatibility of devices from different vendors across the same control system. This is coupled with vendors seeking to provide Modbus capability in their products to meet the demands of their customers, which in turn also leads to strong market competition and the associated cost benefits for customers. These two factors reinforce each other and have led to Modbus being considered the 'de facto standard in multi-vendor integration' (Park et al. 2004).

Due to its nature as a transmission protocol only, Modbus can additionally be converted across multiple physical network types, architectures, and media.

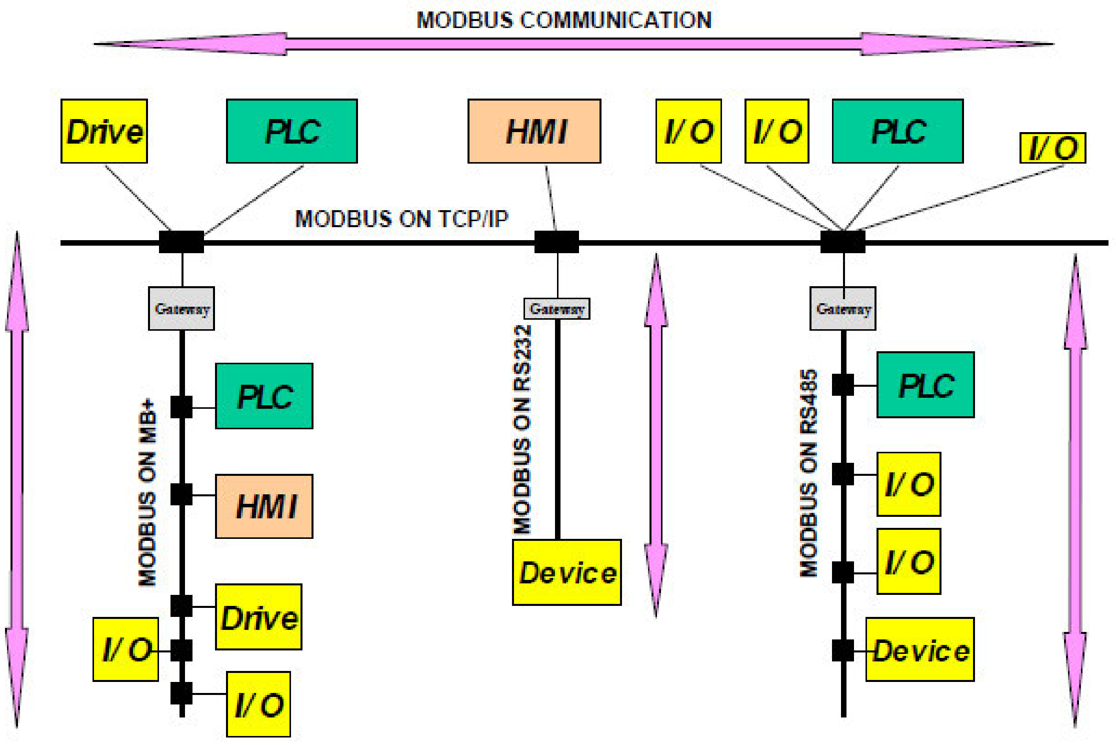


Figure 2.2 Example of MODBUS Network Architecture (Modbus Organisation 2012)

As demonstrated in *Figure 2.2* Modbus transmissions over TCP/IP may not only provide direct communication between a client and server, but rather be converted via appropriate equipment to other physical methods such as serial communications, as part of the same greater network. Inbuilt addressing included in each Modbus transmission designates the destination device. Whilst this functionality will not be directly required for the implementation of this project, the provisions made for it in the Modbus protocol must still be understood and considered. Correct handling of this inbuilt functionality will still be necessary to correctly generate properly formatted Modbus requests, as outlined in Section 2.3 of this chapter.

2.2 Modbus Application Protocol Overview

The Modbus Application Protocol is freely available, and its specification is maintained by the Modbus Organisation. This is fortunate as it allows the operation and requirements of this protocol to be easily assessed and implemented. The primary specifying document in this regard is the Modbus Application Protocol Specification V1.1b3 (2012).

All Modbus transactions consist of a Client device issuing a request to a Server device. The server evaluates the request made and, assuming the request is valid, performs the desired action and sends the required response to the client, as depicted in Figure 2.3 below.

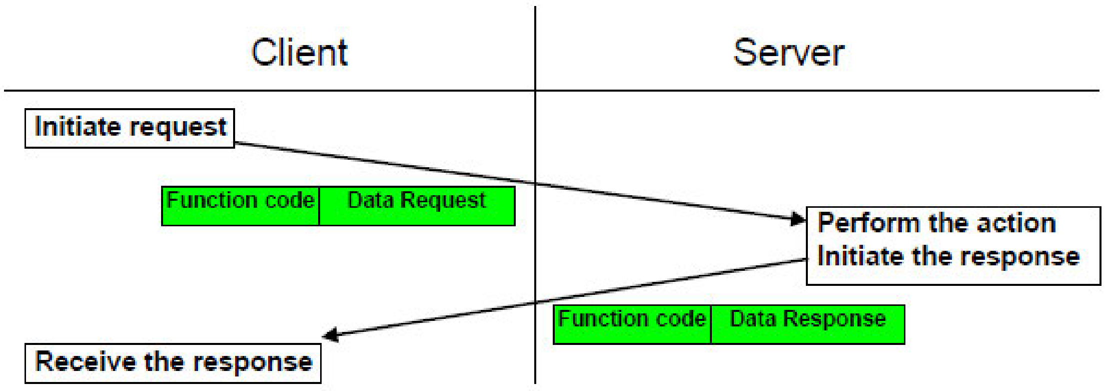


Figure 2.3 Standard Successful Modbus Transaction (Modbus Organisation 2012)

Should the client request prove invalid the server will perform no action and respond to the server with a relevant exception code as shown in Figure 2.4, which allows the client to take further action as required or submit further requests. This process and its specifics are covered further in Section 2.4 of this chapter.

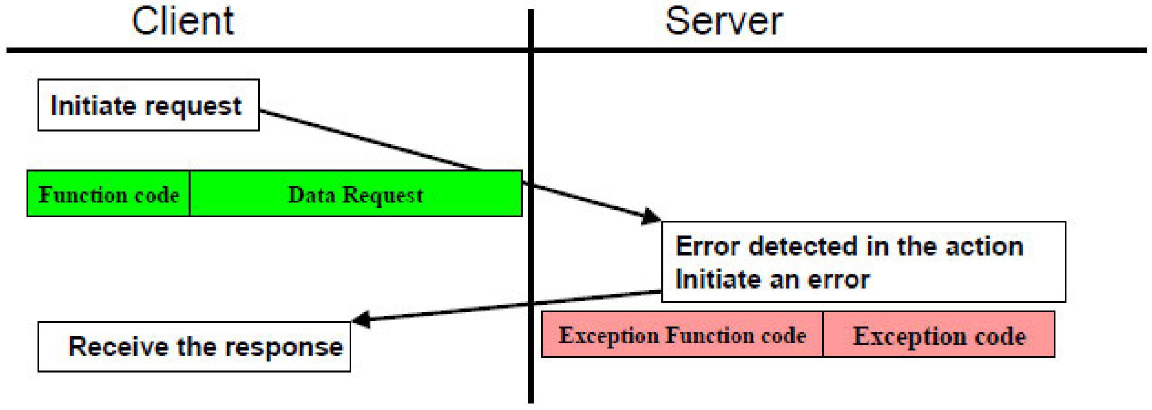


Figure 2.4 Standard Unsuccessful Modbus Transaction (Modbus Organisation 2012)

At its core a Modbus Application Protocol request or response consists of a simple Protocol Data Unit, or PDU, that is independent of whatever the underlying communication layer may be (Modbus Organisation 2015).

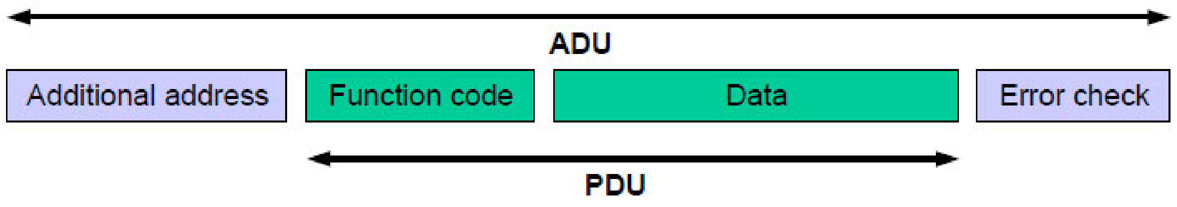
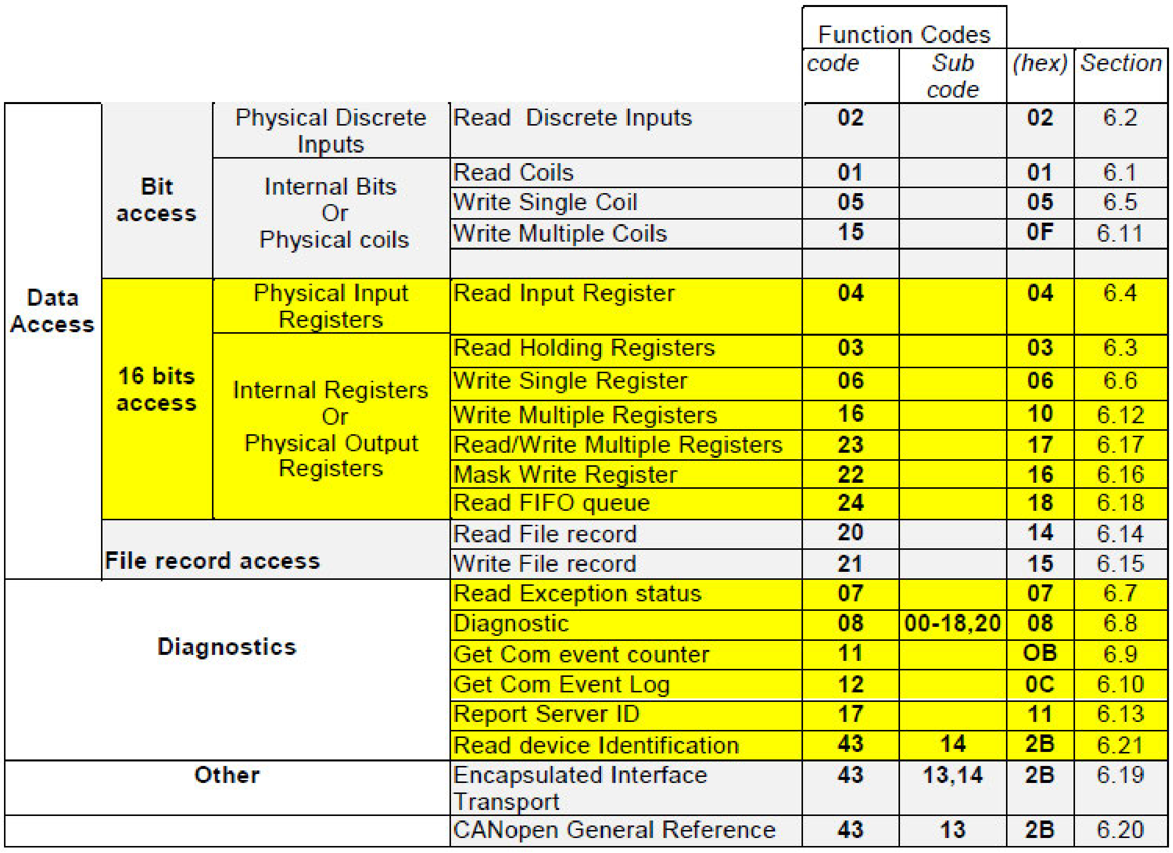


Figure 2.5 General Modbus Frame (Modbus Organisation 2012)

The use of the Modbus protocol on various networks or communication buses may necessitate the addition of additional data preceding or following the Protocol Data Unit in a single transmission. For example, the use of the Modbus application protocol over a serial bus with multiple attached devices would require the addition of a preceding device address, in addition to a following error check value. These additional data fields, in addition to the Protocol Data Unit itself, make up a Modbus Application Data Unit, or ADU, as shown in Figure 2.5 General Modbus Frame (Modbus Organisation 2012).

The format of the Application Data Unit is specific to the particular communication method used, and is specified by relevant secondary documents maintained by the Modbus Organisation. For the purpose of this project only the Modbus TCP/IP format is of relevance, and will be covered in additional detail in Section 2.3 of this chapter.

The Protocol Data Unit consists of Function Code, which specifies the action required, as well as additional data as required by the specified function. The Modbus documentation specifies a set of standard function codes, as shown in Figure 2.6 below, and also makes allowance for the development of user, or vendor, defined function codes. For the purposes of this project, specifically that the constructed interface device be capable of configuration for use with a broad range of devices, the focus will be placed on the use of standard function codes to ensure cross compatibility.



				Function Codes			
				code	*Sub code*	*(hex)*	*Section*
Data Access	**Bit access**	Physical Discrete Inputs	Read Discrete Inputs	**02**		**02**	6.2
		Internal Bits Or Physical coils	Read Coils	**01**		**01**	6.1
			Write Single Coil	**05**		**05**	6.5
			Write Multiple Coils	**15**		**0F**	6.11
	16 bits access	Physical Input Registers	Read Input Register	**04**		**04**	6.4
		Internal Registers Or Physical Output Registers	Read Holding Registers	**03**		**03**	6.3
			Write Single Register	**06**		**06**	6.6
			Write Multiple Registers	**16**		**10**	6.12
			Read/Write Multiple Registers	**23**		**17**	6.17
			Mask Write Register	**22**		**16**	6.16
			Read FIFO queue	**24**		**18**	6.18
	File record access		Read File record	**20**		**14**	6.14
			Write File record	**21**		**15**	6.15
Diagnostics			Read Exception status	**07**		**07**	6.7
			Diagnostic	**08**	**00-18,20**	**08**	6.8
			Get Com event counter	**11**		**OB**	6.9
			Get Com Event Log	**12**		**0C**	6.10
			Report Server ID	**17**		**11**	6.13
			Read device Identification	**43**	**14**	**2B**	6.21
Other			Encapsulated Interface Transport	**43**	**13,14**	**2B**	6.19
			CANopen General Reference	**43**	**13**	**2B**	6.20

Figure 2.6 Standard Modbus Function Codes (Modbus Organisation 2012)

Individual function codes required to be implemented for this project, and their specific data formats, will be covered in additional detail in Section 3.7 in Chapter 3.

2.3 Modbus TCP/IP Ethernet Specifics

Implementing Modbus over TCP/IP introduces several additional elements to the standard Modbus serial implementation, including the possibility of the TCP/IP transmissions being converted onto serial communication buses with multiple attached devices. The nature of TCP/IP communications required that additional care is taken to ensure the correct pairing of requests and their respective responses, as well as ensuring the entire message has been received where it may have been sent is separate packets.

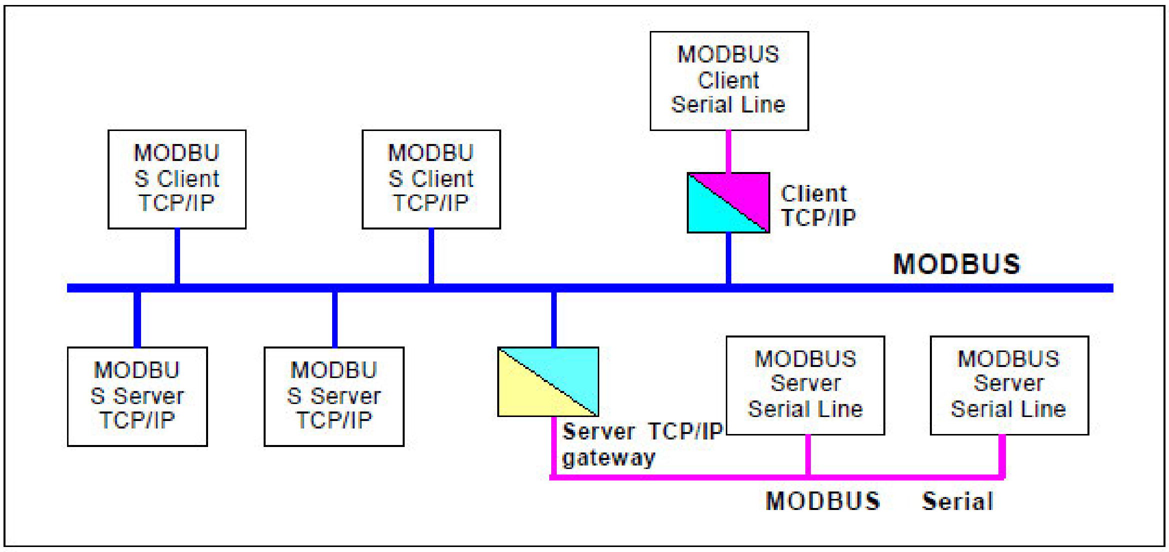


Figure 2.7 Example Modbus TCP/IP Network Layout (Modbus Organisation 2006)

Figure 2.7 illustrates an example of a possible Modbus TCP/IP network. Note that this includes gateway devices intended to convert Modbus requests and responses from TCP/IP to Serial buses and vice versa. For this project the device will be intended to operate solely as a client device communicating with a single server device via direct connection with no additional message routing required.

It should be noted that Modbus TCP connections are directed at TCP Port 502 by default (Modbus Organisation 2006). However other Ports may be used, with more advanced devices often featuring this as a configurable setting. Thus, consideration should be given in the final device code to allow user modification of the Port number to be used, in the event that this is required.

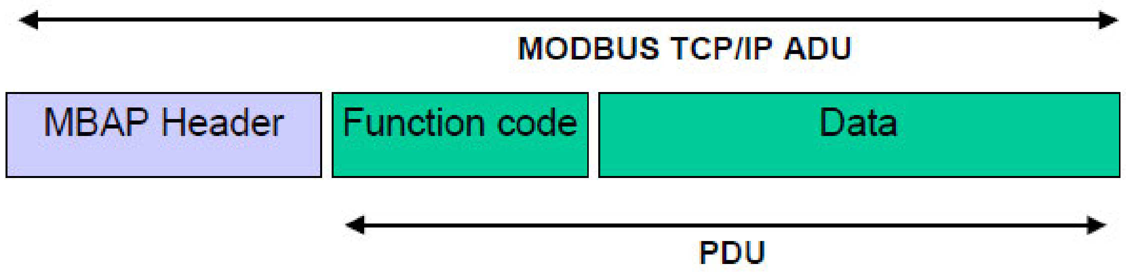


Figure 2.8 Modbus TCP/IP request/response format (Modbus Organisation 2006)

As can be seen in Figure 2.8 a Modbus request or response sent over TCP/IP consists of the Protocol Data Unit, preceded by a Modbus Application Protocol header, or MBAP. Whilst the Protocol Data Unit follows the same format as laid out in the Modbus Application Protocol Specification, the MBAP header is specific to Modbus implemented over TCP/IP and is stated by the Modbus Organisation (2006) to contain the following information:

- Transaction Identifier (2 bytes); A value set by the client when sending the request that is copied directly into the response by the server, to allow positive matching of received responses with their respective requests
- Protocol Identifier (2 bytes); Used for multi-protocol systems, a default value of 0 is used to denote the use of standard Modbus Protocol
- Length (2 bytes); Denotes the total byte count of the request/response following the final byte of the Length value itself, including the Unit Identifier and the entire PDU. Used to ensure the entire request/response has been received in instances where the Application Data Unit may have been separated into separate packets for transmission over TCP/IP
- Unit Identifier (1 byte); Used to specify the sub address of the intended device in the event the request is intended to be converted to a serial bus with multiple attached devices

As the device to be constructed intends to operate solely on a single client, single server basis, with no conversion to or from serial line communications, the unit identifier can be effectively set to a zero value and removed from consideration. Similarly, as no cross-protocol operation is required, the Protocol Identifier can also be fixed at a zero value. Transaction Identifier values should be automatically managed by the final device software and require no direct user input or consideration. Likewise, generation of the length field should be processed automatically, and response length fields used to automatically manage the segregation of individual responses received.

2.4 Modbus Error Handling

A standard Modbus request is processed by the receiving server as laid out in Figure 2.9.

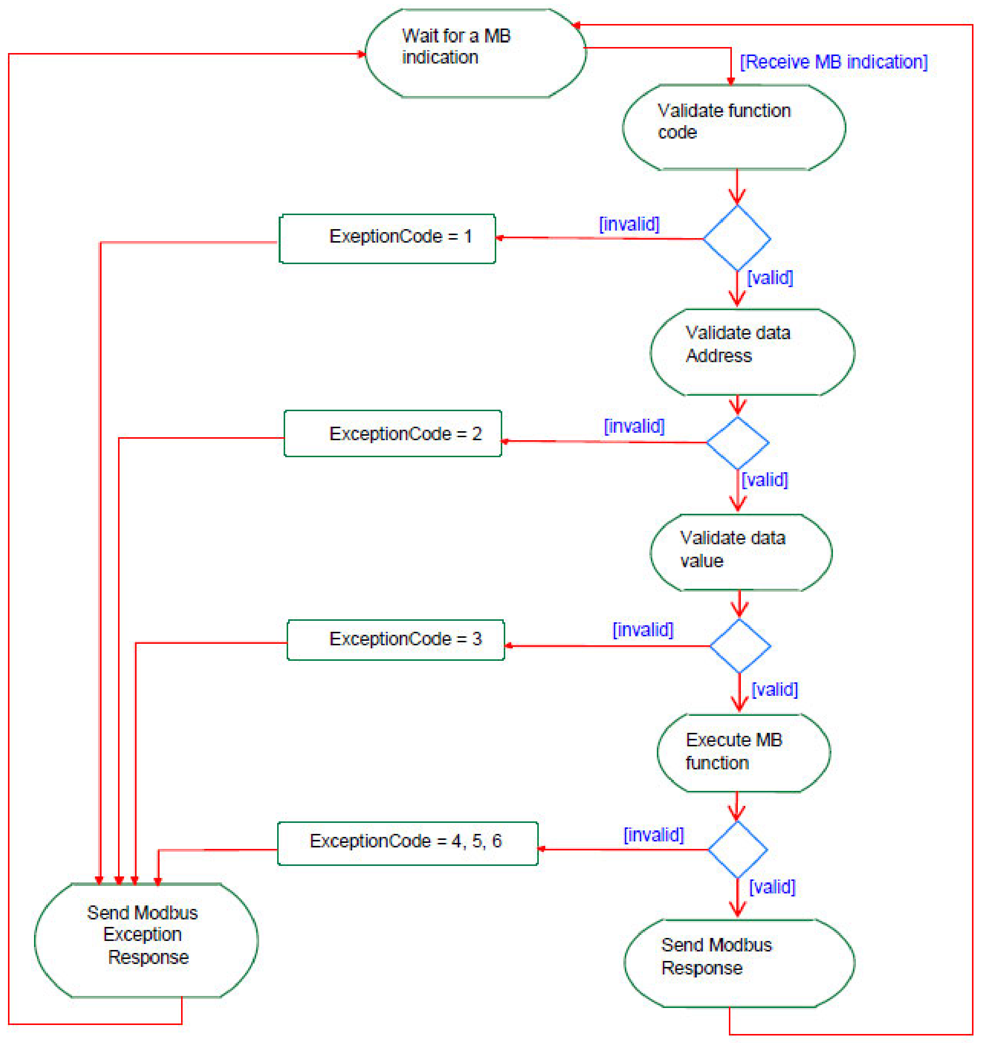


Figure 2.9 Standard Modbus transaction (Modbus Organisation 2012)

As can be seen the server will evaluate the Function Code, Data Address, and Data value in turn and raise an exception code of 1, 2, or 3 respectively if any are found to be invalid values. Assuming all three segments are valid the server will proceed to execute the requested function or may return an exception code 4 to indicate that the server device has failed. In the event that the server anticipates the request will take a long time to process it may return exception code 5, to acknowledge that the server is processing the request but anticipates a delay before returning a response. This allows the client to refrain from considering the request timed out, and await the eventual response. Should a server receive a request whilst already process a long duration request it will issue an exception code 6 to indicate that it is presently busy and unable to process further requests.

In the event that any exception code is raised, a response is sent to the client with the Function Code set to the sum of the initial request function code and 0h80, and the data segment set to the exception code raised. The configuration device to be constructed will be required to accept these exception responses and interpret them to determine the correct action to be taken.

2.5 Existing Modbus Communications Options

As part of this project a review was performed of presently available hardware or software aimed at performing similar functions. Several software packages were identified that provide functionality to manually issue Modbus requests and interpret responses from a standard laptop or other computer. However, these software packages provided only individual request/response functionality, with no mechanism for the automatic generation and execution of multiple requests/responses to perform a desired task, such as full configuration of a Modbus device. Additionally, the requirement of standard laptop or computer hardware is incompatible with one of the primary intentions of this project, namely to be a low cost and highly portable unit. For these reasons it has been concluded that the intended purpose of this project is sufficiently novel that no significant commercial effort has been expended towards the production of a tool to fulfil it. This necessitates that this project proceeds based upon the base Modbus TCP specification and guiding documentation in order to fulfil its stated goals.

2.6 SEL-710 Motor Protection Relay

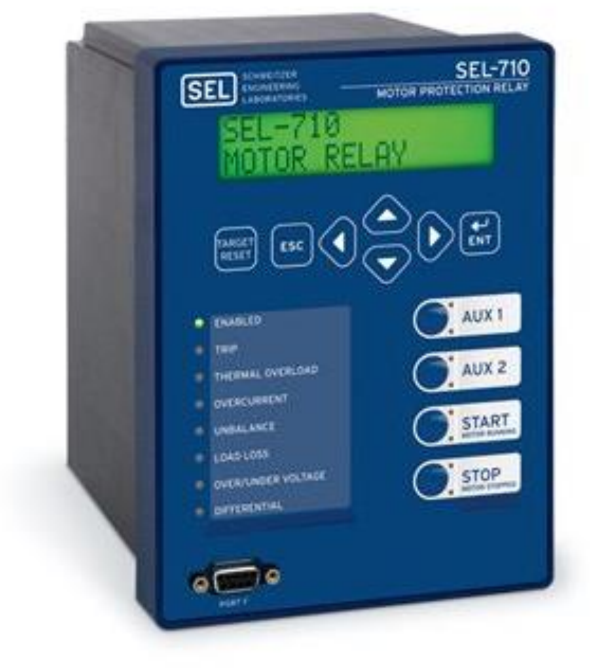


Figure 2.10 The SEL-710 Motor Protection Relay (Schweitzer Engineering Laboratories 2015)

The SEL-710 is a Motor Protection Relay, or MPR, designed and marketed by Schweitzer Engineering Laboratories. Many different configurations of the relay are available, allowing the customer to pick specific additional functions for particular installations. However, the base model still consists of a full range of features, including Overload, Locked Rotor, Phase current imbalance, Short Circuit, and other protection functions as well as motor monitoring functions (Schweitzer Engineering Laboratories 2015).

Due to the high number of configuration options and protective functions relays such as the SEL-710 are often selected for use with large high cost motors or for critical systems which warrant the added expense. In these instances, downtime due to initial commissioning delays at install or during fault diagnosis situations, can incur significant financial penalties. Additionally, these processes often require personnel with significant experience to complete quickly and with minimal disruption.

This particular model of relay provides the option of configuration via either manufacturer specific proprietary software or manually using the basic user interface. The manual user interface consists of a 2 by 16-character display, enter, escape, and basic directional keys. This significantly restricts the speed at which parameters can be configured, especially in the case of numerical values where each digit must be incrementally selected and entered. The possibility of a setting being entered incorrectly, and the potentially high consequences of such an error, necessitates manual checking of each entered value for positive confirmation, further increasing the task duration.

The relay includes basic fault diagnosis functions, including recording of recent trip causes with a limited snapshot of data for each function. However more advanced logging functions are not available via the user interface for more advanced diagnosis of intermittent faults, and no capability exist for logging of specific values over a short to medium duration, or at any significant frequency.

Individual configurable relay parameters, as well as read only status values are mapped to predetermined Modbus addresses. This necessitates that the configuration tool software must possess the ability to correctly cross reference given parameter names to their relevant Modbus addresses.

3 Analysis of Hardware and Software Requirements

3.1 Consideration of Consequential Effects

The most significant potential consequential effects of this Modbus TCP configuration tool would be the potential to expose unqualified individuals to live Low voltage electrical circuits. Whilst the test unit is an extra low voltage powered relay, there is the potential that this device could be used in situations where Low Voltage supplies are present on the Modbus device to be interfaced with. In this instance the work should only be performed by suitably qualified individuals who are aware of the potential hazards and how to remove or mitigate them before interacting with the device. This is beyond the ability of the configuration tool to control; however, it is pertinent that a suitable warning be included in the user instruction or program prompts, and perhaps the device casing itself. This warning should state that the configuration device should only be used by suitably qualified personnel, and the dangers of exposure to live Low Voltage circuits should always be considered when working on electrical equipment of any sort.

A secondary potential consequence is that a failure of the device to correctly configure a Modbus device could cause incorrect operation of said equipment. For this reason, a prompt will be included in the user instructions, and possibly code prompts, to advise the user to only user the device on offline equipment, and to manually confirm all key parameters are set correctly before returning the equipment to service.

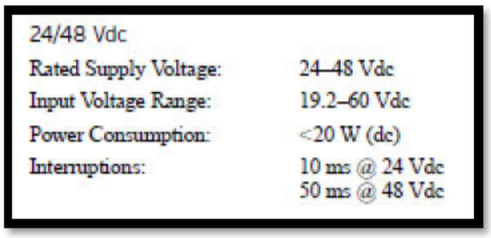
Consideration was given to the use of others intellectual property throughout this project. Due to the nature of the project heavy use of the Modbus standard and guiding documentation has been necessary. This use is permitted by the Modbus Organisation (Modbus Organisation 2020), along with limited reproduction of materials for the purposes of study and research. Additionally, several third-party software packages were utilized, and in these instances the relevant licensing documentation was reviewed to confirm that its use for the purpose of this project was permitted.

It is further noted that the configuration tool to be constructed for this project will be a prototype for the purposes of research, and is not intended to be immediately used in real world environments in such a state. Significant additional work, and a much broader range of considerations, would be required for a device required to be used as a finished commercial product.

3.2 SEL-710 Hardware Considerations

For the purposes of this project the SEL-710 test device is not intended to be connected to an actual installation, due to the risk of spurious operation during prototyping and testing. Rather the SEL-710 will be assembled into a separate test assembly that provides supply power to the relay, and allows access to the Modbus Ethernet port and manual user interface for device testing purposes.

An extra low voltage direct current model has been selected as opposed to a standard 230VAC low voltage model. This is intended to remove the significant electrical hazard of a 230VAC direct low voltage supply being required. In turn this raises the requirement of providing an appropriate extra low voltage DC power supply to supply the test unit.



24/48 Vdc

Rated Supply Voltage:	24–48 Vdc
Input Voltage Range:	19.2–60 Vdc
Power Consumption:	<20 W (dc)
Interruptions:	10 ms @ 24 Vdc 50 ms @ 48 Vdc

Figure 3.1 SEL-710 ELV Model power supply requirements (Schweitzer Engineering Laboratories 2015)

Data presented in Figure 3.1 documents the power supply requirements of the SEL-710 that must be provided by the test assembly. A 24 VDC supply voltage is selected due to the ready availability of supplies in this voltage. This supply voltage in conjunction with the stated power consumption of <20W and be used to calculate the required supply current will be ~0.83 A as per Equation 3.1. For practical purposes a 1 Amp power supply can be used.

$$P = VI$$

$$I = \frac{20}{24}$$

$$I = 0.83\dot{3}\,A$$

Equation 3.1 Test unit current requirement

Consideration must also be given to protection of the electrical supply for instances of overload or short circuit

Figure 3.2 designates terminals A01 and A02 as the positive and negative supply terminals respectively, as well as showing the location of these terminals and Modbus TCP/IP Port 1.

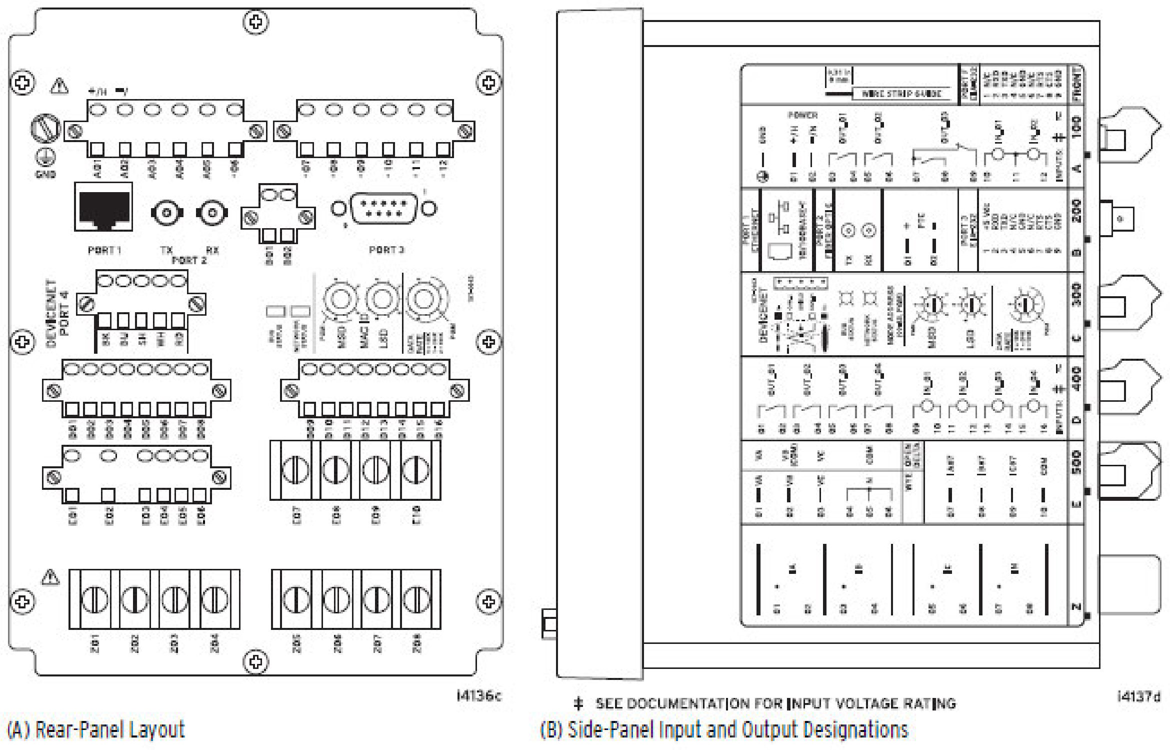

i4136c
(A) Rear-Panel Layout
‡ SEE DOCUMENTATION FOR INPUT VOLTAGE RATING
i4137d
(B) Side-Panel Input and Output Designations

Figure 3.2 SEL-710 Terminal arrangement and assignment (Schweitzer Engineering Laboratories 2015)

3.3 Hardware Requirements

In line with the stated objective of this project, and the literature review that has been completed, the following hardware requirements have been identified.

Low cost; directly addressing the project objective for the end design to be accepted for use in field situations where it could be exposed to damage, without significant financial burden.

Compact; in order to keep the device at a physical size and weight that allows its ergonomic use as a handheld device for prolonged periods.

Processor and RAM; sufficient to perform constructing and interpreting the required Modbus requests and responses, as well as basic graphical and user interface functions.

An inbuilt RJ45 Ethernet port and associated hardware/support; to allow direct connection to Modbus TCP equipment without the added cost of added adaption equipment.

Availability of TCP/IP level software libraries; to allow ease of use of these functions by the Modbus application level software to be developed.

Ability to access external memory storage; such as an SD card or USB flash drive, in order to access user configuration and data files, or store output files.

Support for graphical user interface and touch input; in order to allow the final device to leverage the availability of small low-cost HDMI/USB touch displays to create a more intuitive user interface. This would require a single USB port and HDMI output port for ease of connection, with native software support.

These determined requirements are assigned priorities as shown in Table 3.1 below.

Requirement	**Priority**
Cost	2 (Med)
Dimensions	2 (Med)
Processor Speed	2 (Med)
RAM	2 (Med)
RJ45 Ethernet Port	3 (High)
TCP/IP Support	3 (High)
External Memory Support	3 (High)
Graphical User Interface Capability	1 (Low)
Touch Input Capability	1 (Low)

Table 3.1 Requirement Priorities

3.4 Available Hardware

As one of the primary objectives of this project is to create a low-cost device from readily available hardware platforms the following product options were considered:

- Arduino Uno Microcontroller Platform
- Raspberry Pi Model 3B+ Single Board Computer
- BeagleBone Black Single Board Computer

3.4.1 Arduino Uno

Figure 3.3 Arduino Uno (Arduino AG 2020)

The Arduino Uno is a Microcontroller based prototyping platform based on an Atmel ATmega328P (Arduino AG 2020). This option was rapidly discarded due to its lack of native graphics support or on-board RJ45 Ethernet Port. An additional hardware module, the Arduino Ethernet Shield, exists to provide the required hardware to enable Ethernet communications. The requirement of this additional hardware renders this option significantly less cost effective when compared to the other options considered, for no discernible gain. Additionally, when considering this projects' optional goal of producing a graphical user interface for the final configuration tool, it was found that the options are significantly more limited for this platform when compared with the other options considered.

3.4.2 Raspberry Pi 3B+

Figure 3.4 Raspberry Pi 3B+ (Raspberry Pi Foundation 2020)

The Raspberry Pi Model 3B+ is a single board computer produced by the Raspberry Pi Foundation. It is based on a System-on-Chip ("SoC") produced by Broadcom, that contains most of the components found in a standard computer within a single integrated circuit (Raspberry Pi Foundation 2020). This allows the Raspberry Pi to be produced as an exceptionally compact unit, and for low cost, whilst still maintaining the ability to perform the functions of an ordinary computer. Additionally, the 3B+ model includes integrated hardware for USB and RJ-45 Ethernet connectivity. Various compatible operating systems are available, with the default "Raspbian" operating system being produced and supplied by the Raspberry Pi Foundation. Many commercial products are available to add compact graphical touch interfaces to this board, with the Raspbian operating system providing native touch interface support. No integrated memory storage is present, and must be provided by an additional SD card.

3.4.3 BeagleBone Black

Figure 3.5 BeagleBone Black (BeagleBoard.org Foundation 2019)

The BeagleBone Black is a single board prototyping platform based on an ARM Cortex-A8 processor (BeagleBoard.org Foundation 2019). It integrates several more advanced features than the Raspberry Pi, such as on-board flash memory, additional low-level input/output pins, and specific processor and graphics capabilities. None of these features were determined to add a discernible advantage for the purposes of this project. Integrated USB and RJ-45 Ethernet hardware are provided. Multiple c compatible operating system options exist, with the most popular options being Linux based (BeagleBoard.org Foundation 2019).

3.5 Hardware Platform Selection

The following comparison of the Raspberry Pi 3B+ and BeagleBone Black was performed as shown in Table 3.2. This was collated using information from the Raspberry Pi Foundation (2020), BeagleBoard.org Foundation (2019), and Element14 (2020).

	Cost	Dimensions (mm)	Processor Speed (GHz)	RAM	RJ45 Ethernet Port	TCP/IP Support	External Memory Support	Graphical User Interface Capability	Touch Input Capability	Python Capable	Total Score
Raspberry Pi 3B+ (RPI)	$56.63	85x56	1.4	1GB	YES	YES	YES	YES	YES	YES	
BeagleBone Black (BBB)	$86.34	87x54	1	512MB	YES	YES	YES	YES	YES	YES	
Preference	RPI	BBB	RPI	RPI	TIE	TIE	TIE	TIE	TIE	TIE	
Weight	2 (Med)	2 (Med)	2 (Med)	2 (Med)	3 (High)	3 (High)	3 (High)	1 (Low)	1 (Low)	2 (Med)	
RPI Score	2	0	2	2	3	3	3	1	1	2	19
BBB Score	0	2	0	0	3	3	3	1	1	2	15

Table 3.2 Comparison of Hardware Platforms

Based on this comparison the Raspberry Pi 3B+ was selected as the chosen hardware platform.

3.6 SEL-710 Software Considerations

Figure 3.6 lists the Modbus Function codes supported by the SEL-710 relay, for consideration when selecting the Function codes to be supported by this configuration device.

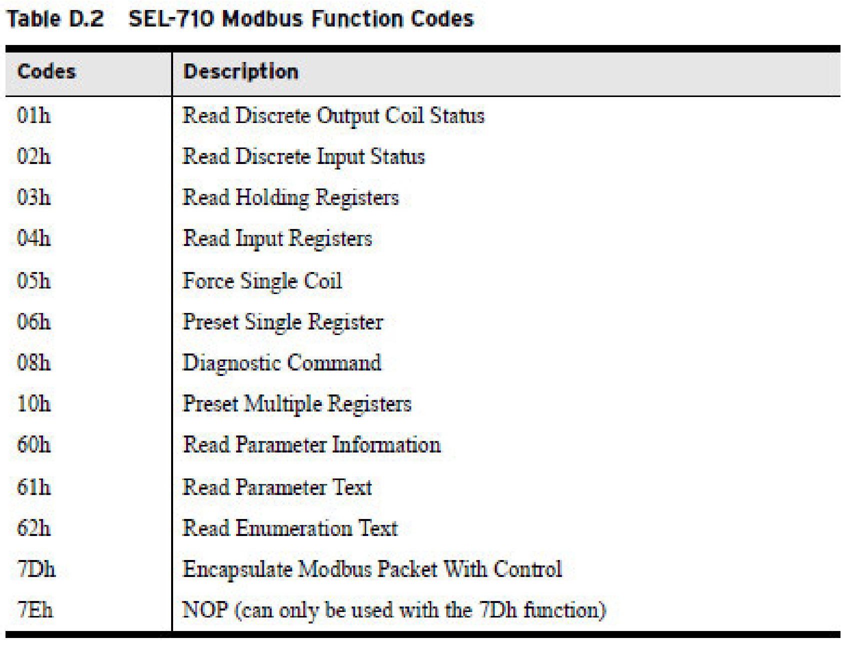


Table D.2 SEL-710 Modbus Function Codes

Codes	Description
01h	Read Discrete Output Coil Status
02h	Read Discrete Input Status
03h	Read Holding Registers
04h	Read Input Registers
05h	Force Single Coil
06h	Preset Single Register
08h	Diagnostic Command
10h	Preset Multiple Registers
60h	Read Parameter Information
61h	Read Parameter Text
62h	Read Enumeration Text
7Dh	Encapsulate Modbus Packet With Control
7Eh	NOP (can only be used with the 7Dh function)

Figure 3.6 SEL-710 Accepted Modbus Function Codes (Schweitzer Engineering Laboratories 2015)

3.7 Required Modbus Function Codes

The following Modbus standard Function Codes have been selected for implementation in this project. This selection enables various modification and monitoring methods for the parameters and I/O of the device to be configured. All of these Function Codes are additionally supported by the selected SEL-710 test device. Not all of these functions will be directly required to perform the stated configuration and interrogation functions. However, their implementation at this stage will increase the versatility of software package and its ability to be adapted perform additional tasks as may be desired by users.

- **0h01**: Read Discrete Output Coil (Bit) Status
- **0h02**: Read Discrete Input Status
- **0h03**: Read Holding Registers
- **0h04**: Read Input Registers
- **0h05**: Force Single Coil (Bit)
- **0h06**: Preset Single Register
- **0h10**: Preset Multiple Registers

These function codes are fully defined in the Modbus specification, allowing them to be implemented in the configuration device code with confidence.

3.8 Software Requirements

With the selection of the Raspberry Pi 3B+, the corresponding Raspbian Operating system was selected for this project. This decision was based upon the ready availability of this option, its native support of graphical touch interface hardware, and the abundance of available documentation and support.

The core of the configuration device operation required production of client software that is capable of initiating a TCP connection with the test unit server, forming and sending a correct Modbus request, receiving and interpreting the response, and maintaining or closing the TCP connection as required. Additionally, exception code handling, parameter data input/output into external files, and a method of being configured for use with different Modbus devices was required.

To this end Python 3.6 was selected as the programming language the software is to be developed in, due its inclusion of several libraries capable of handing the low-level functions required such as TCP connections, and file I/O (Python Software Foundation 2020). Additionally, there are several options available for graphical user interfaces that are supported by Python, which provided further benefit for this projects' optional goal.

Initial code was developed to manually "proof of concept" test each of the selected Function Code. Once this core code was developed it was tested and verified using the SEL-710 local keypad and screen interface. Additionally, testing was performed on the final code using guidance set out in the Modbus Conformance Test Specification for Modbus TCP (Modbus Organisation 2009), with the process suitably modified for a client type device rather than a server type.

The operation of the core software capability to be developed for this project is essentially that of a Modbus TCP Client device. Guidance on the operation of such devices is given in the supporting Modbus documentation as shown in Figure 3.7. Execution of such operation requires several constituent software functions to be developed. Specifically, these must include TCP management functions, Modbus Request generation, Modbus Response processing, and Error handling/interpretation.

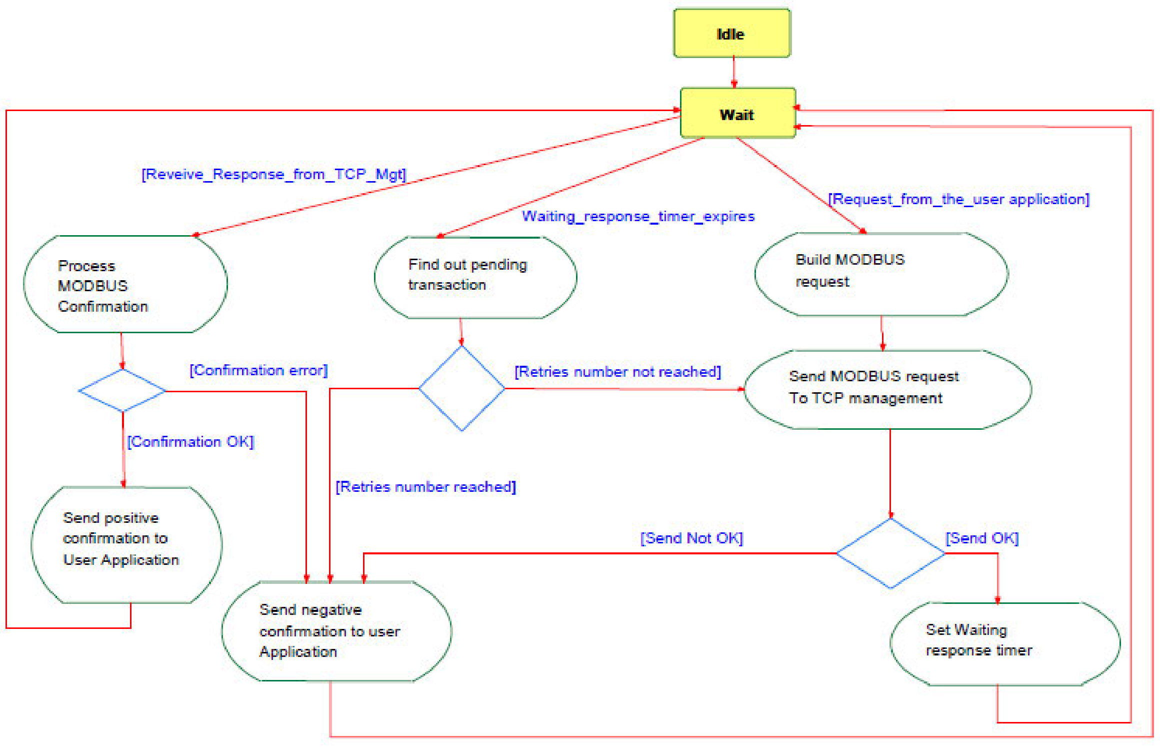

Idle
Wait
[Reveive_Response_from_TCP_Mgt]
Waiting_response_timer_expires
[Request_from_the_user application]
Process MODBUS Confirmation
Find out pending transaction
Build MODBUS request
[Confirmation error]
[Retries number not reached]
Send MODBUS request To TCP management
[Confirmation OK]
[Retries number reached]
Send positive confirmation to User Application
[Send Not OK]
[Send OK]
Send negative confirmation to user Application
Set Waiting response timer

Figure 3.7 Modbus Client Activity Diagram (Modbus Organisation 2006)

With the core Modbus TCP client code and its constituent functions successfully developed, appropriate application code is required to utilise this core software to perform the desired Read and Write operations. There are two primary automated functions intended to be performed by this tool. First is a "Configuration" function, which would write values to multiple registers as specified by a parameter settings file input by the user. Second is an "Interrogation" function, which perform either a single read of a small set of registers, or multiple reads at a given frequency for a set duration, the data obtained from which would be written to a log file for diagnostics and fault-finding purposes.

Further additional desired features require several auxiliary code functions to allow efficient user input and output. Firstly, file operations to import and parse data from a device "Configuration" file would be required. The intent of this file is to provide information to the software package pertaining to the Modbus addresses specific to the device to be configured that correspond with individual parameter names. Additional data for each parameter should also be provided by this file, including its scaling, format, units, read/write status, minimum setting, maximum setting, and default setting. Additionally, for parameter that use an enumeration or bit enumeration format, details of the individual enumeration settings should be included. As these files will need to be generated for each model of Modbus device intended to be interfaced with, both YAML and XML Markup languages were considered for use. Both of these languages were determined to be capable of the required task, namely that a set of individual properties could be set for each individual parameter entry. Additionally, both languages are both human and machine readable, and Python software packages exist that allow ease of import and export. YAML was ultimately selected due to its more compact and simplified structure and symbol set, with the intent that this will simplify the generation of additional configuration files when needed. The arrangement of data within this file is detailed in Chapter 5, specifically Section 5.2.

Further file operations for the importation of register settings from the user provided parameter settings file during a configuration operation, and for the export of obtained data to a log file during and interrogation operation. Comma Separated Value files will be utilised for both the parameter settings and log files, due to their suitability for the information to be contained. Additionally, this format is readily compatible with many commercial spreadsheet programs, allow ease of production and interpretation of the data held therein by the end user. The arrangement of data within this file is detailed in Chapter 5, specifically Section 5.3 and Section 5.4 respectively.

Upon completion and successful testing of these individual code modules they were then integrated together into a cohesive whole under a unified user interface, presented as individual scripts for each of the primary functions, Configuration and Interrogation.

Finally, with the code completed the configuration tool was assembled, the final code loaded, and put through full testing of all of its functions from the end user standpoint. Additionally, testing was performed on the final code using guidance set out in the Modbus Conformance Test Specification for Modbus TCP (Modbus Organisation 2009), with the process suitably modified for a client type device rather than a server type.

4 Hardware Design

4.1 SEL-710 Test Unit Design

For use in the development of this software a test unit has been constructed as per Figure 6.1 to allow safe and easy interaction with the SEL-710. A generic 24 VDC "Plug Pack" Power supply with a rating of 1.25A and integral overload and short circuit protection was selected and obtained for the purposes of powering this test unit.

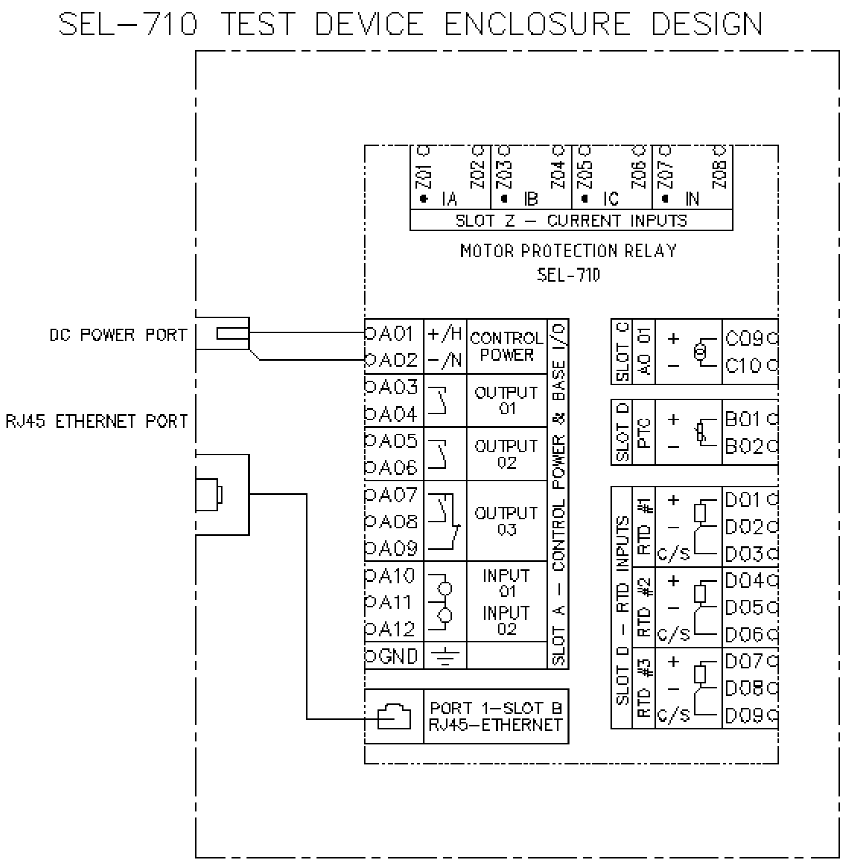


Figure 4.1 SEL-710 Test unit schematic

4.2 Configuration Tool Functional Design

The configuration device prototype assembly was designed as per the schematic shown in Figure 6.2. During software development the Raspberry Pi was directly connected to the SEL-710 test unit, with direct power supply omitting the battery power module. The intent is that the tool is able to import and export parameter and configuration data files from and external USB flash drive via the included external USB port. Likewise, the external RJ-45 and Micro USB ports enable connection to a Modbus device or power supply without any disassembly of the unit itself. The final assembled prototype is shown in Figure 4.3.

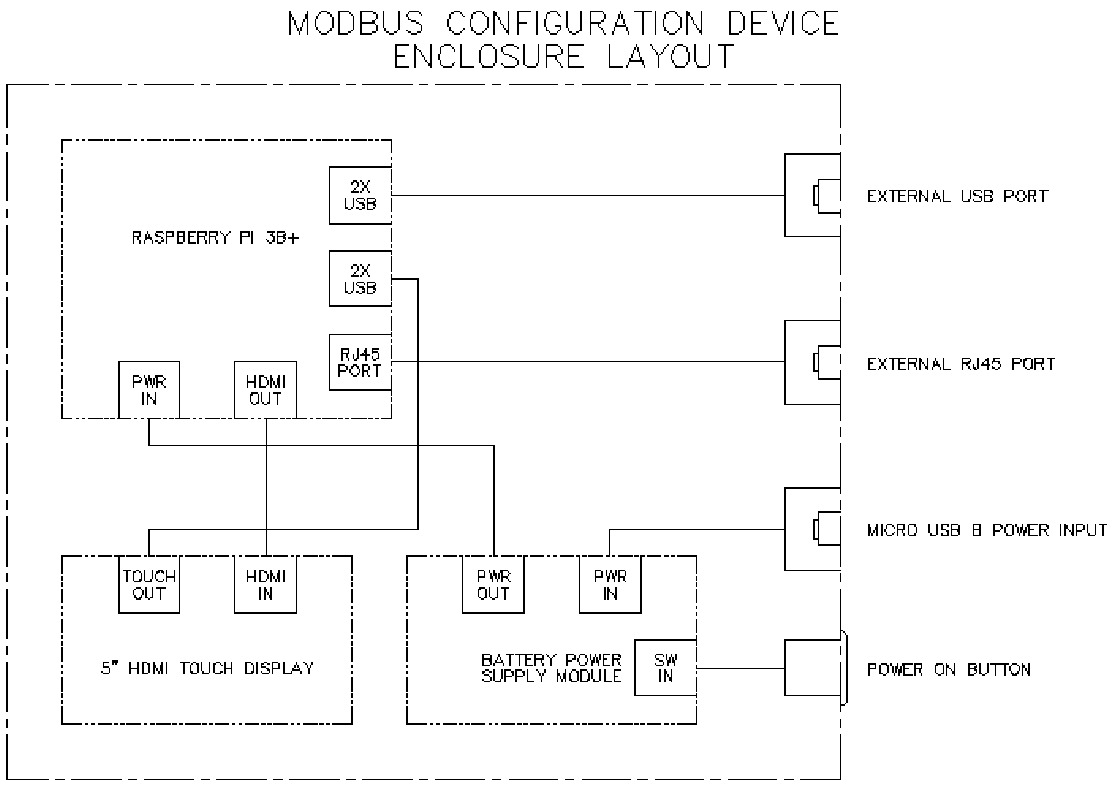


Figure 4.2 Modbus Configuration Device Schematic

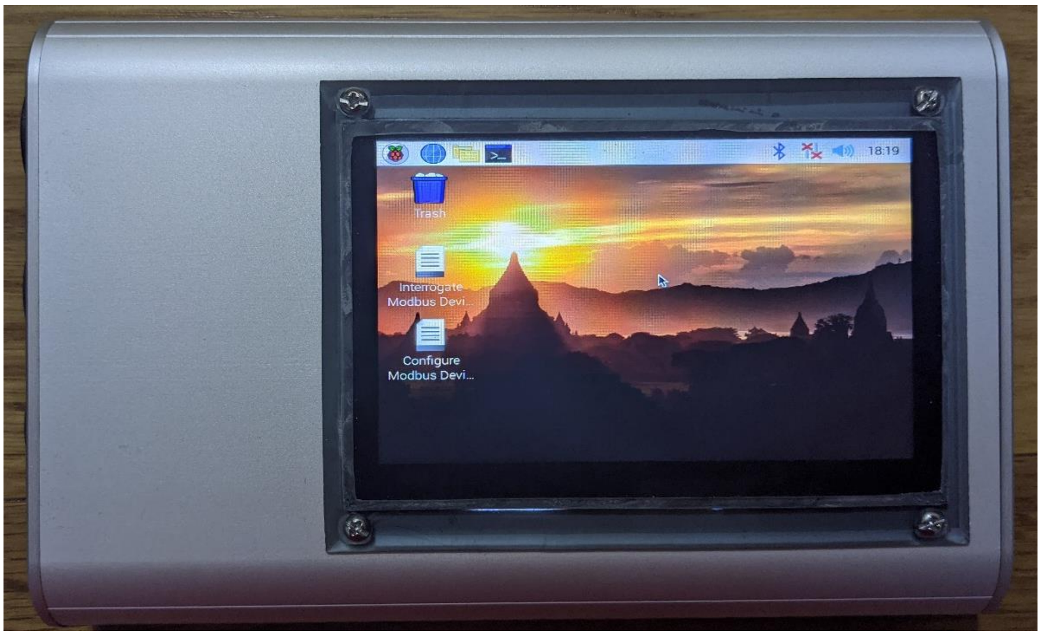


Figure 4.3 Assembled Prototype Hardware

4.3 Raspberry Pi 3B+

Figure 4.4 Raspberry Pi Model 3B

The process that led to the selection of the Raspberry Pi Model 3B+ as the primary prototyping component for this project is detailed in Chapter 3, specifically Section 3.5. In addition to the base Raspberry Pi Board, a 32 GB Class 10 SD card was obtained to serve as the primary memory device for this platform. This decision was based upon the manufacturer's recommendations for optimal performance (Raspberry Pi Foundation 2020). This SD card was subsequently loaded with the Raspbian desktop operating system as provided by the Raspberry Pi Foundation.

4.4 HDMI Touch Display

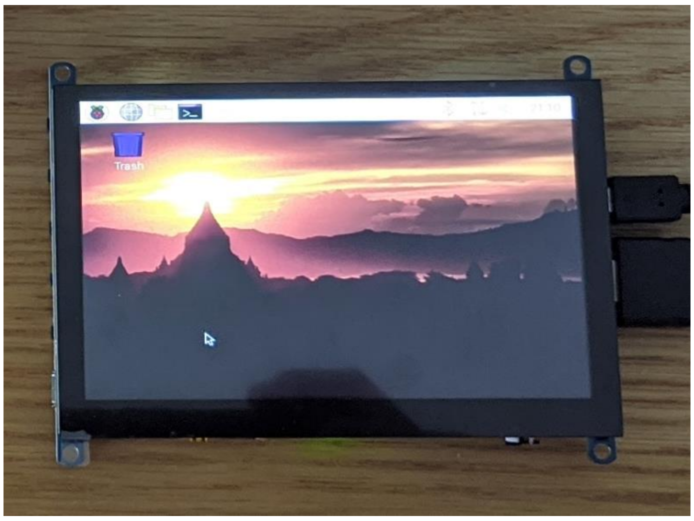


Figure 4.5 Waveshare 5-Inch Touchscreen Display

The criteria selected for the user interface display were determined based upon the following logic.

- Compatibility with HDMI video input; Due to the Raspberry Pi 3B+ native hardware support for HDMI video output, this was deemed the optimal option to meet the low-cost goal of the configuration tool. Simultaneously the reduction of unnecessary additional components potentially removed additional points of failure, thus improving the overall robustness of the final design.
- USB compatible power supply; Due to the USB power supply already present for the Raspberry Pi, conformance with this requirement removes the need for additional power supply components specific to the display.
- 5-inch display size; This physical dimension was found experimentally to be large enough for convenient use, whilst remaining compact enough in size to be readily integrated into a portable device.
- USB compatible touch interface; the availability of multiple USB ports on the Raspberry Pi board made this the optimal interface for use by the touch input interface.

A large number of commercially available options were found to meet these requirements, with only minimal distinguishing features to aid in selection of an individual unit. The decision was made to select the Waveshare 5 Inch HDMI LCD Capacitive touch display as shown in Figure 4.5. This was primarily due to its availability at time of purchase, comparably low cost, build quality, and native support for its touch interface by the Raspbian operating system.

4.5 Battery Power Supply

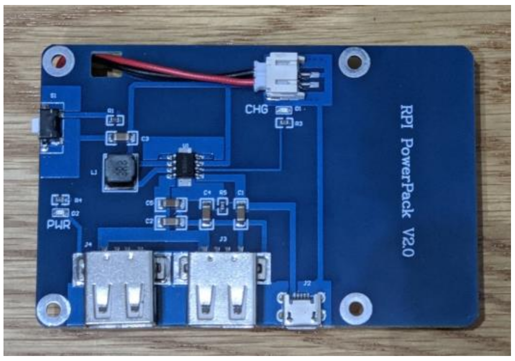


Figure 4.6 Battery Power Supply Module

A wide variety of battery power supplies compatible with the Raspberry Pi board are commercially available. However, at the time of construction, disruptions to global supply chains significantly limited the availability of these modules. Due to the auxiliary nature of this component to the overall project, and the uncertain lead times of any ordered units, the primary criteria for the selection of this component became availability. This consideration, in conjunction with the requirement that the unit be capable of supplying both the Raspberry Pi 3B+ and the user interface display simultaneously limited the options to a single generic unit available locally and in stock, as shown in Figure 4.6. The board features USB power outputs, a micro USB input for charging, and an integrated power switch. The power switch may be manually removed and replaced with soldered leads to allow attachment of the external power button of the prototype configuration tool. The integrated battery is a 3.7 V 3800 mAh lithium ion cell. Battery life testing of the assemble unit was performed to determine the practicality of this battery capacity during use, with the target being in excess of one hour of operation.

4.6 Enclosure

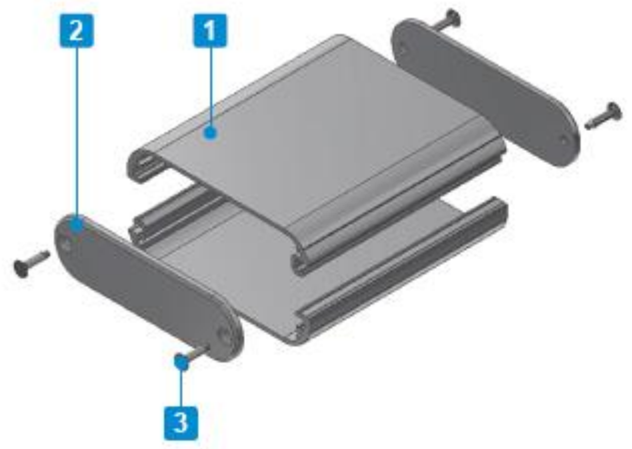


Figure 4.7 Takachi MXA Series Enclosure (Takachi Electronics Enclosure Co. Ltd. 2020)

The Takachi MXA4-13-20SSP enclosure was selected to house the configuration tool hardware. This decision was based upon its robust aluminium construction, ready availability at the time of sourcing, and low cost. These factors were deemed to make it suitable for meeting the goals of the final configuration tool. Additionally, the modular and easy to disassemble design simplifies the process of machining external entry points for the user interface and hardware ports. This factor proved advantageous for the prototype nature of the design.

4.7 Bill of Materials

The following lists detail the hardware resources determined to be required for this project. Other minor components were utilised at stages, but these were not critical to project progression, being mostly consumables and for convenience purposes.

SEL-710 Test Unit:

- SEL-710 Motor Protection Relay, PN: 071002BBB9X0X831211
- 24VDC, 1 Amp, Plug pack power supply with integrated overload and short circuit protection
- Enclosure to suit SEL-710
- 2.5mm DC power connection port, panel mount, with wire tails for connection
- Ethernet connection port, panel mount, with internal cable tail for connection
- 3m Cat 5 Ethernet patch cable

Configuration Tool Unit:

- Raspberry Pi, Model 3B+
- Raspberry Pi Plug pack power supply, Micro USB B output
- Micro SD card, 32GB, Class 10
- External power switch, with internal wire tails for connection, dust resistant
- Micro USB B connection port, panel mount, with internal cable tail for connection, dust resistant
- Ethernet connection port, panel mount, with internal cable tail for connection, dust resistant
- Waveshare 5” HDMI Monitor, with USB touch interface
- Enclosure to suit Raspberry Pi, touch panel, and associated equipment
- HDMI flexible ribbon style cable, with 2x HDMI connectors
- USB 2.0 flexible ribbon style cable, with 2x Micro USB 2.0 connectors
- Rechargeable USB power supply, 3.7V 3800mAh battery

5 Software Design

5.1 Overall Structure

As determined in Chapter 3, the software package developed is written in the Python 3 programming language, specifically Python version 3.6, and run on the Raspbian desktop operating system. It is important to note that Python is an interpreted programming language as opposed to compiled language such as C. This effectively means that the produced Python code is directly “interpreted” by the Python “interpreter” software during execution, as opposed to compiled language where the written code is first compiled into an executable file before execution (Python Software Foundation 2020).

Python code can be separated into separate files, or modules, and subsequently imported for use during execution of other Python files (Python Software Foundation 2020). This allows the code to be conveniently separated into separate files, with individually defined functions imported into the executed files, thus keeping individual code segments concise and easier to interpret.

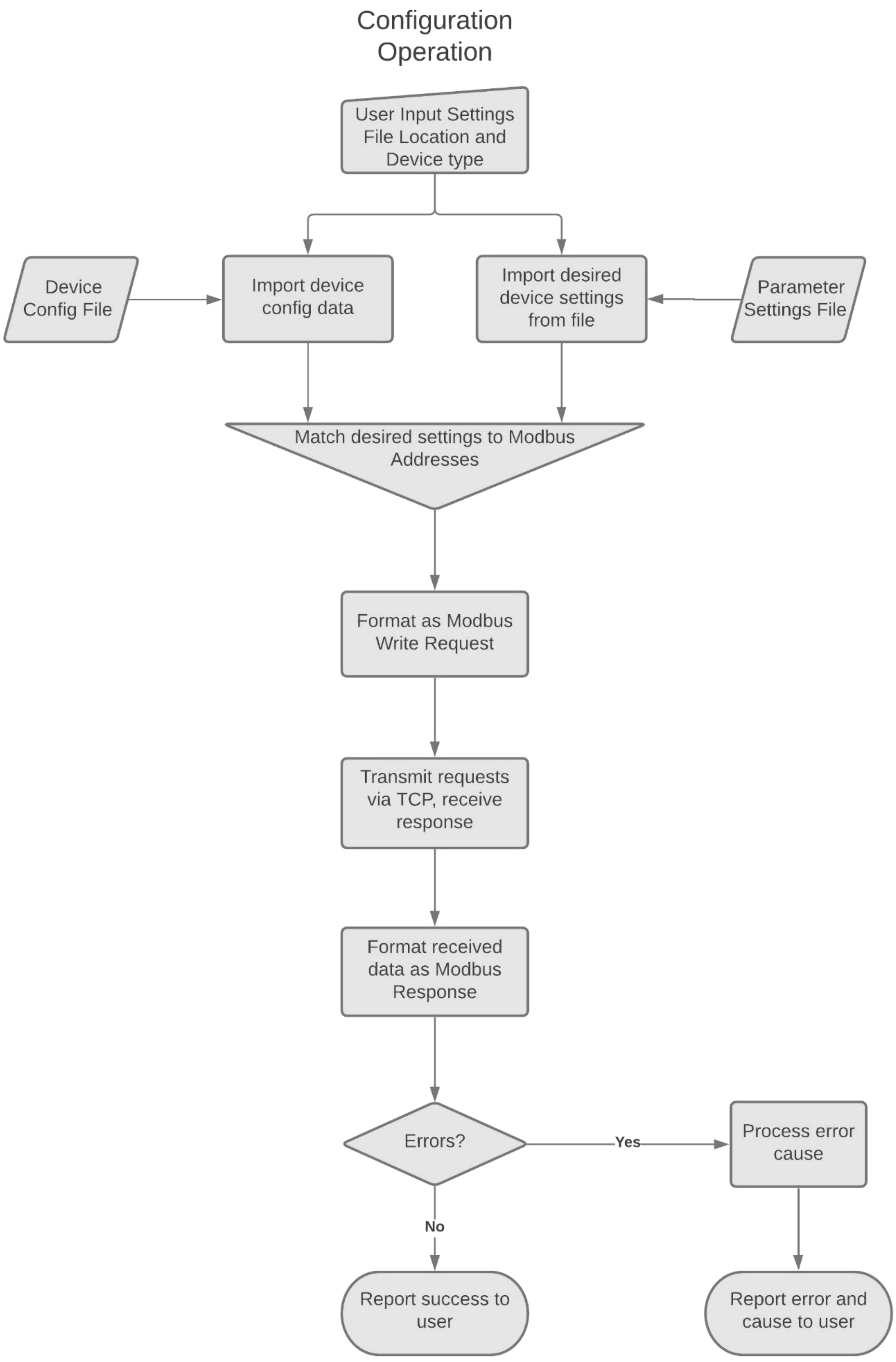
For the purposes of this this project the following modules included with the standard Python interpreter:

- “socket”, software library for managing and automation of TCP communications
- “select”, software library used for determining state and timeout of TCP connection
- “yaml”, software library for the import and export of YAML formatted files
- “csv”, software library for the import and export of CSV formatted files
- “time”, software library for basic timekeeping functions

The required formats of the individual input and output files are defined in Section 5.2 through to Section 5.4 of this Chapter. The final software package is separated into individual modules as defined in Section 5.5 through to Section 5.13 of this Chapter. The sequence of operation for overall Configuration and Interrogation operations are shown by Figure 5.1 and Figure 5.2 below, respectively. It should be noted that whilst the downloading of a parameter set from a Modbus TCP device is completely possible given the software modules developed, no automated routine was created for this function. This was due to the intended purpose of the configuration element of this device, which was intended to be used in an industrial setting. In such a setting the preferred source of configuration data should always be obtained through approved master data sources, not extracted from field devices which may or may not have been modified by third parties, unknown to the user. For this reason such functionality was not provided in an automated routine, but could be performed manually by use of the software functions defined in this chapter, specifically those aimed at read operations on multiple holding registers.

These software modules, specifically those concerning TCP communications, were designed to default to a target device IP address of 192.168.1.2/24. Allowance is made in the user interface to modify this address, but will additionally require modification of the Raspberry Pi's network configuration. Details on the initial setting of this configuration, as required for correct operation of these modules, are detailed in Section 5.14 of this chapter.

A set of user instructions for the final prototype are provided in Appendix F.



Configuration Operation
User Input Settings File Location and Device type
Device Config File
Import device config data
Import desired device settings from file
Parameter Settings File
Match desired settings to Modbus Addresses
Format as Modbus Write Request
Transmit requests via TCP, receive response
Format received data as Modbus Response
Errors?
Yes
Process error cause
No
Report success to user
Report error and cause to user

Figure 5.1 Configuration Function Flowchart

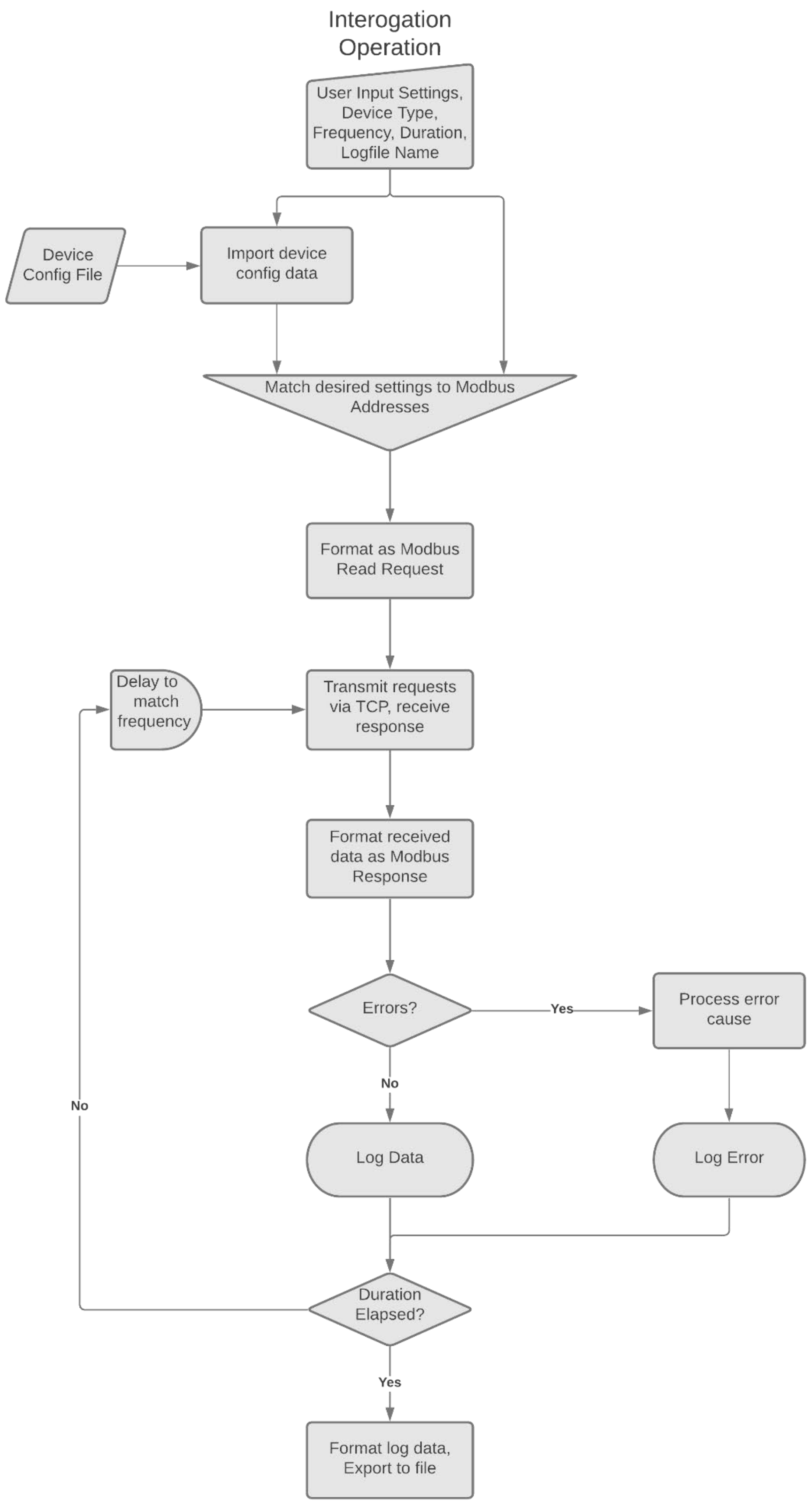

Interogation Operation
User Input Settings, Device Type, Frequency, Duration, Logfile Name
Device Config File
Import device config data
Match desired settings to Modbus Addresses
Format as Modbus Read Request
Delay to match frequency
Transmit requests via TCP, receive response
Format received data as Modbus Response
Errors?
Yes
Process error cause
No
No
Log Data
Log Error
Duration Elapsed?
Yes
Format log data, Export to file

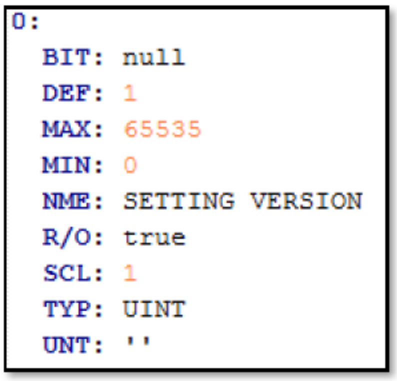
Figure 5.2 Interrogation Function Flowchart

5.2 Device Configuration File Format

The Device Configuration file utilises the YAML language to store the relevant device parameter data. An individual entry is made for each parameter number, with each entry containing a data structure that specifies the relevant details for that parameter. The parameter details contained within each entry, and their respective field names, are show in Table 5.1**Error! Reference source not found.** below. An example of a standard entry is shown in Figure 5.3.

Parameter Detail	Field Key
Parameter Name	NME
Read Only flag	R/W
Unit	UNT
Minimum Value	MIN
Maximum Value	MAX
Default Value	DEF
Scale Factor	SCL
Bit Descriptor List, if applicable	BIT

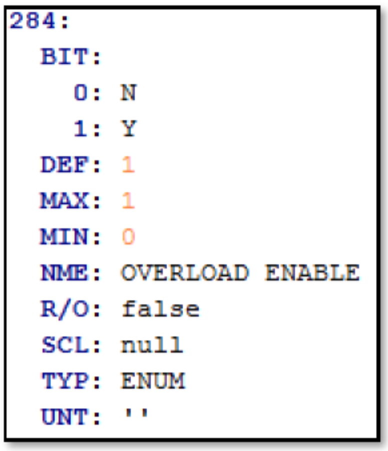
Table 5.1 Configuration File Entry Fields



```
0:
  BIT: null
  DEF: 1
  MAX: 65535
  MIN: 0
  NME: SETTING VERSION
  R/O: true
  SCL: 1
  TYP: UINT
  UNT: ''
```

Figure 5.3 Standard Configuration File Entry

Additionally, for Enumeration and Bit Enumeration parameters, the Bit Descriptor field contains an additional list specifying each available setting. An example entry using the Bit Descriptor field is shown in Figure 5.4

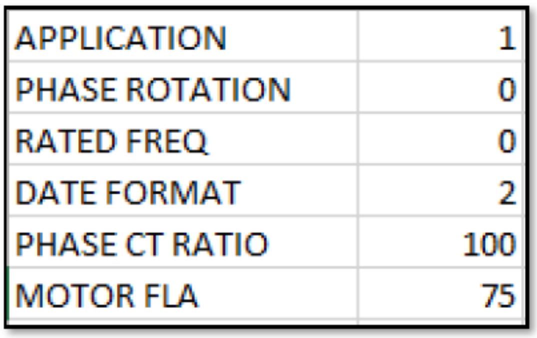


```
284:
  BIT:
    0: N
    1: Y
  DEF: 1
  MAX: 1
  MIN: 0
  NME: OVERLOAD ENABLE
  R/O: false
  SCL: null
  TYP: ENUM
  UNT: ''
```

Figure 5.4 Bit Descriptor Configuration File Entry

5.3 Parameter Settings File Format

The Parameter Settings file uses a standard Comma Separated Value file format to store the parameter/setting pairs. An entry is made for each parameter value that is desired to be set. The first value in each entry is the name of the parameter to be set, as shown in the device configuration file. The second value in each entry is the desired setting for the specified parameter. For parameters that are scaled the unscaled value is entered; the scale factor as specified by the device configuration file is applied during the configuration routine. An excerpt of a settings file containing several entries is shown in Figure 5.5.

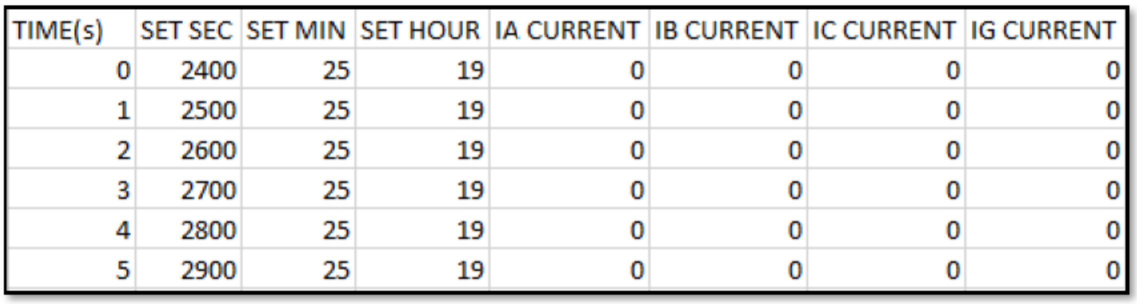


APPLICATION	1
PHASE ROTATION	0
RATED FREQ	0
DATE FORMAT	2
PHASE CT RATIO	100
MOTOR FLA	75

Figure 5.5 Settings File Entries

5.4 Interrogation Log File Format

The Interrogation Log file uses a standard Comma Separated Value file format to store the interrogated values in an accessible fashion. The initial entry is made to label each column with its respective parameter name. An additional entry is then made for each sample taken, consisting of a time elapsed value in seconds, followed by the scaled values of each parameters value. An excerpt of a log file containing several entries is shown in Figure 5.6.



TIME(s)	SET SEC	SET MIN	SET HOUR	IA CURRENT	IB CURRENT	IC CURRENT	IG CURRENT
0	2400	25	19	0	0	0	0
1	2500	25	19	0	0	0	0
2	2600	25	19	0	0	0	0
3	2700	25	19	0	0	0	0
4	2800	25	19	0	0	0	0
5	2900	25	19	0	0	0	0

Figure 5.6 Log File Entries

5.5 File Input/Output

This module performs the functions required to import data from the Parameter Settings and Device Configuration files. Additionally, a function is provided to export collected data to a Log file. This module consists of file “MDB_File_IO.py”, included in Appendix B.1.

5.5.1 Import Device Configuration (Read_Config)

This function accepts a string variable, which specifies the directory and name of the required Device Configuration file. It then utilizes the inbuilt Python open() function to open the specified file in a read-only mode. The YAML format data held therein is then parsed using the yaml.load() function provided by the imported “yaml” module and returned as a variable consisting of nested Python “dictionary” data types to maintain the data structure. A Python dictionary data types consists of a set of “name”:”value” pairs (Python Software Foundation 2020). In this instance a “nested” dictionary refers to additional dictionary variables being assigned as the values of a higher-level dictionary, to provide a multi-tiered data structure. This format allows the data corresponding to individual Modbus addresses specific to the device the file describes. Finally, the inbuilt close() function is automatically invoked to close the open file.

5.5.2 Import Parameter Settings (Read_Settings)

This function accepts a string variable, provided by the user, which specifies the directory and name of the specified Parameter settings file. It then utilizes the inbuilt Python open() function to open the specified file in a read-only mode. The CSV format data held therein is then parsed using the csv.reader() function provided by the imported “csv” module and then returned as a single level dictionary data type matching the parameter names with their desired values. Finally, the inbuilt close() function is automatically invoked to close the open file.

5.5.3 Export Log File (Write_Log)

This function accepts a string variable, input by the user, which specifies the desired directory and name of the Log File to be output containing the data contained within a two-level nested list variable passed to the function. In this instance each entry is intended to contain a list of values, with the first element specifying sample timestamps, and each additional element corresponding with a device register value that was read. This function utilizes the inbuilt Python open() function to open the specified file in a create/write/overwrite mode. The provided dictionary variable is then parsed using the csv.writer() function form the imported “csv” module and subsequently written to the open log file. This format allows the data obtained via the output comma separated variable file for review in any compatible software, including many commercially available spreadsheet programs. Finally, the inbuilt close() function is automatically invoked to close the open file.

5.6 Modbus Address Encoding/Decoding

This module performs the functions required to cross reference parameter names with their respective entries in the device configuration file. Two function are provided, namely “Addr_Settings” and “Addr_Lookup”, to perform functions as defined below. This module consists of file “MDB_Addr.py”, included in Appendix B.2.

5.6.1 Addr_Settings

This function accepts two input arguments. The first input is the desired settings data dictionary variable, as returned by the “Read_Settings” function from the given settings file. The second input is the configuration data dictionary variable, as returned by the “Read_Config” function from the device configuration file. From the configuration data the function generates a cross reference dictionary that allows the parameter Modbus address to be immediately determined when given the parameter name. The function then processes each variable in the given settings data into an output dictionary variable in turn. During processing the parameters read/write status is checked, if the parameter is found to be read-only an error is raised and program execution ended. This will only occur if the given settings file contained an erroneous entry to write a value to a read-only parameter. The desired setting for the parameter is then tested to ensure it is between the minimum and maximum allowable values for that parameter. If so, an error is again raised and program execution cancelled. Again, this should only occur if an error in the given settings file results in the specification of an incorrect value for the parameter. If the value is within the acceptable range, it is scaled according to its specified scaling factor, converted into a binary bytes format, and stored in the output dictionary variable entry that corresponds to its parameter address. This output dictionary variable is then returned by the function upon completion.

5.6.2 Addr_Lookup

This function accepts the configuration data dictionary variable as returned by the "Read_Config" function from the device configuration file as its sole input argument. From this data it generates a cross reference dictionary with an entry for each parameter name. Each entry in this dictionary consists of a list containing the parameter address, and a Boolean value that corresponds with the parameters read-only status. Thus, the second element of a read-only parameter entry will be "True". This cross-reference dictionary is then returned by the function upon completion.

5.7 Modbus Format

This module does not contain any standalone function, but rather a single python "class" named "MBusPkt". The class is essentially a data type with functions inbuilt, allowing the value/format it returns to be strictly specified. The inbuilt functions can also be invoked externally, causing the class to take the given input and perform the function internally. In this instance the purpose of the class is to contain a single Modbus TCP Application Data Unit, which in turn contains an MBAP header and Protocol Data Unit. The operation of the class and its internal functions are individually specified below. This module consists of file "MDB_Format.py", included in Appendix B.3.

5.7.1 __init__

This internal function is automatically invoked when a variable of the “MBusPkt” type is created. When created the class may be given a single input argument consisting of a Modbus TCP ADU in binary bytes format.

If this input is given, this function checks that the input is in binary bytes format, and if so that it is at least eight bytes in length. The “Length” data is then extracted from the packet, and checked to be consistent with the length of the relevant packet segment. If any of these conditions are not met an error message is raised and program execution is ended, as the packet is not a valid ADU. If the packet meets these requirements it is separated into its component fields which are stored internally along with the fully composed ADU.

If no input is given, this function sets its internal PDU fields as blank, and MBAP header fields from set default values, with the length field automatically calculated.

5.7.2 __repr__

This function simply determines how the class represents itself in string format if called. In this instance it returns the value of its internal ADU packet variable in hexadecimal format.

5.7.3 __bytes__

This function simply determines how the class represents itself in binary bytes format if called. In this instance it returns the value of its internal ADU packet, which is already stored as binary bytes internally.

5.7.4 set_PKT

This function performs the same as the “__init__” function if it received an initial input ADU. This is to allow setting of the full ADU for a variable of this class after it has been initialised.

5.7.5 set_TX_ID

This function accepts a transaction identifier value in integer format as its sole input argument. It checks the given input is the correct format, raising an error and ending program execution if it is not. It then converts the given value to binary bytes format of the correct two-byte length and stores it in its internal transaction identifier field. It then updates its internal length field, and recomposes its internal ADU packet with the updated component fields.

5.7.6 set_PRTCL

This function accepts a Protocol Identifier value in integer format as its sole input argument. It checks the given input is the correct format, raising an error and ending program execution if it is not. It then converts the given value to binary bytes format of the correct two-byte length and stores it in its internal Protocol Identifier field. It then updates its internal length field, and recomposes its internal ADU packet with the updated component fields.

5.7.7 set_UN_ID

This function accepts a Unit Identifier value in integer format as its sole input argument. It checks the given input is the correct format, raising an error and ending program execution if it is not. It then converts the given value to binary bytes format of the correct one-byte length and stores it in its internal Unit Identifier field. It then updates its internal length field, and recomposes its internal ADU packet with the updated component fields.

5.7.8 set_FN_CD

This function accepts a Modbus Function Code in integer format as its sole input argument. It checks the given input is the correct format, raising an error and ending program execution if it is not. It then converts the given value to binary bytes format of the correct length and stores it in its internal Function Code field. It then updates its internal length field, and recomposes its internal ADU packet with the updated component fields.

5.7.9 set_DATA

This function accepts the data segment of a Modbus TCP transaction, formatted as a series of binary bytes, as its sole input argument. It checks the given input is the correct format, raising an error and ending program execution if it is not. It stores the given input in its internal Data field, updates its internal length field, and recomposes its internal ADU packet with the updated component fields.

5.8 TCP Communications

This module primarily provides a function that performs the transmission of a list of Modbus TCP Application Data Units over a TCP connection to a Modbus TCP device as requests, and receives the responses given. An additional function handles the generation of a list of ADU requests to be sent from a given list of Protocol Data Units. The exact operation of the two functions, "Comm" and "Gen_Requests", is explained further below. This module consists of file "MDB_Comms.py", included in Appendix B.4.

5.8.1 Comm

This function requires a list of formatted ADU requests as its sole mandatory input argument. Two additional optional input arguments can be accepted to specify the device IP address and TCP port number. If these either of these optional arguments is not given its default value is used in its stead, namely 192.168.1.2 and 502 for the IP address and TCP port respectively. The function opens a TCP connected to the Modbus TCP device using the connection parameters specified then processes each request in turn. Processing the request consists of sending the request ADU to the device, and waiting for a response to be received. If the response is not received within one second the request is considered to have timed out, in which case the connection is closed, an error raised, and program execution ended. Once received the response is read through to the length field of its MBAP header, and then the remaining number of bytes specified by the length field are read. The response is then stored as an entry in a responses dictionary variable corresponding to its transaction identifier, as a "MBusPkt" class variable. Once all requests are processed, the TCP connection is closed. Upon completion the responses dictionary is returned by this function.

5.8.2 Gen_Requests

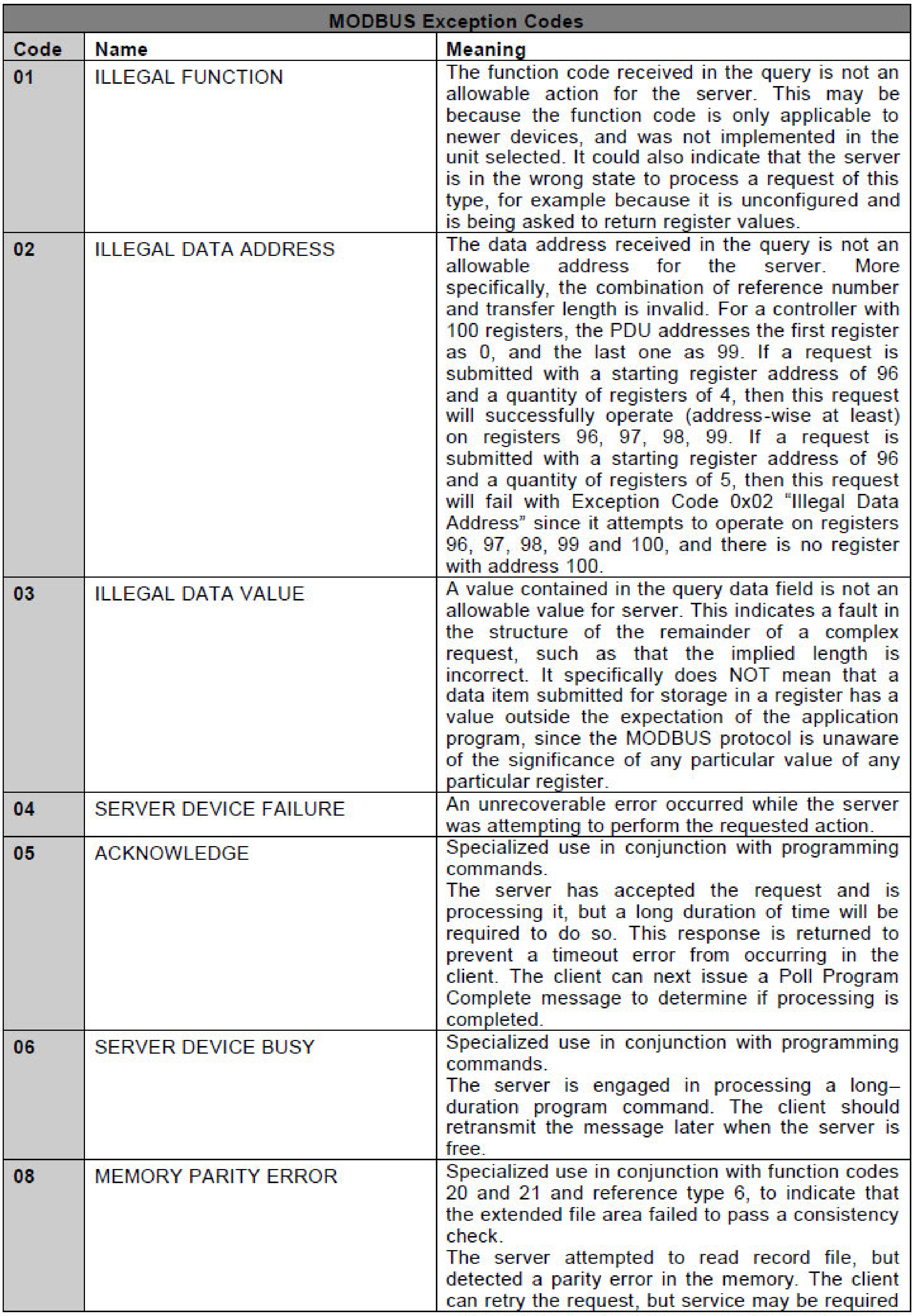
This function accepts a list of PDU's as its sole input argument. Each element of this input list consists itself of list of two values, with the first element being the PDU Function Code in integer format, and the second element being the PDU data payload formatted as a series of binary bytes. For each PDU in the input list a "MBusPkt" class variable is created with its Function Code and Data fields set to those specified. The Transaction Identifier is also set for each variable, with a value of zero for the first PDU, and increasing by one for each subsequent PDU. These request variables are stored in order in a requests list variable, which is returned by the function upon completion.

5.9 Modbus Error Handling

This module provides a single function to interpret the error code contained within the data field of a Modbus TCP transaction that resulted in an error. A dictionary of error response messages corresponding to the error codes is also included, based upon the Modbus specified exception codes shown in Table 5.2. Two additional exception codes specific to Modbus Gateways were excluded as they are not relevant to the function of this tool. The exact operation of the function, “Modbus_Error”, is explained below. This module consists of file “MDB_Error.py”, included in Appendix B.5.

5.9.1 Modbus_Error

This relatively simple function accepts a single Modbus response ADU as its sole input argument. From this ADU it extracts the data field containing the error code. This code is then used in conjunction with the included error message dictionary to select the error message specific to that code. The selected error message text string is then returned by the function upon completion.



MODBUS Exception Codes		
Code	**Name**	**Meaning**
01	ILLEGAL FUNCTION	The function code received in the query is not an allowable action for the server. This may be because the function code is only applicable to newer devices, and was not implemented in the unit selected. It could also indicate that the server is in the wrong state to process a request of this type, for example because it is unconfigured and is being asked to return register values.
02	ILLEGAL DATA ADDRESS	The data address received in the query is not an allowable address for the server. More specifically, the combination of reference number and transfer length is invalid. For a controller with 100 registers, the PDU addresses the first register as 0, and the last one as 99. If a request is submitted with a starting register address of 96 and a quantity of registers of 4, then this request will successfully operate (address-wise at least) on registers 96, 97, 98, 99. If a request is submitted with a starting register address of 96 and a quantity of registers of 5, then this request will fail with Exception Code 0x02 "Illegal Data Address" since it attempts to operate on registers 96, 97, 98, 99 and 100, and there is no register with address 100.
03	ILLEGAL DATA VALUE	A value contained in the query data field is not an allowable value for server. This indicates a fault in the structure of the remainder of a complex request, such as that the implied length is incorrect. It specifically does NOT mean that a data item submitted for storage in a register has a value outside the expectation of the application program, since the MODBUS protocol is unaware of the significance of any particular value of any particular register.
04	SERVER DEVICE FAILURE	An unrecoverable error occurred while the server was attempting to perform the requested action.
05	ACKNOWLEDGE	Specialized use in conjunction with programming commands. The server has accepted the request and is processing it, but a long duration of time will be required to do so. This response is returned to prevent a timeout error from occurring in the client. The client can next issue a Poll Program Complete message to determine if processing is completed.
06	SERVER DEVICE BUSY	Specialized use in conjunction with programming commands. The server is engaged in processing a long–duration program command. The client should retransmit the message later when the server is free.
08	MEMORY PARITY ERROR	Specialized use in conjunction with function codes 20 and 21 and reference type 6, to indicate that the extended file area failed to pass a consistency check. The server attempted to read record file, but detected a parity error in the memory. The client can retry the request, but service may be required

Table 5.2 Modbus Exception Codes (Modbus Organisation 2012)

5.10 Modbus Read request/response handling

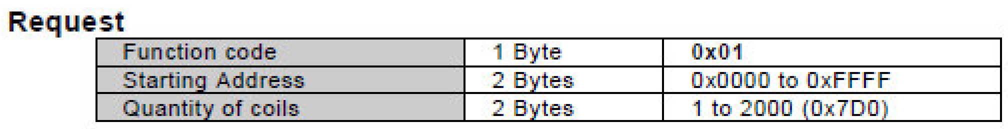
This module provides an array of functions that perform the generation of request Protocol Data Unit details, and handling of response Protocol Data Units for the following Function Codes:

- **0h01**: Read Discrete Output Coil (Bit) Status
- **0h02**: Read Discrete Input Status
- **0h03**: Read Holding Registers
- **0h04**: Read Input Registers

The operation of the included functions, is explained further below. This module consists of file "MDB_Read.py", included in Appendix B.6.

5.10.1 Gen_Read_Discrete_Output_Coil_PDUS (Function Code 0h01)

The purpose of this function is to generate the PDU/s for the execution of Modbus Function Code 0h01 requests, as specified in Figure 5.7. The function accepts a list of Modbus coil addresses, formatted as integers, as its sole input argument. Initially this input argument is checked to be of a list format, with an error code raised and code execution ended if it is not. A sorted list variable is then generated from the given input, in order from lowest address value to the highest. The individual addresses in the sorted list variable are then processed in order. Whilst being processed each address is checked to be of the correct integer format, and within the allowable range for this Modbus function, namely between 0h0000 and 0hFFFF. If either of these conditions is not met an error is raised and program execution is ended. The addresses are then separated into sets that meet the requirement of this Modbus function that each set does not span a range of greater than 2000 subsequent coil addresses. These address sets are each formatted as list variables, and are stored in order as elements of and Address Index list variable. The starting address and number of subsequent coils required to be read to encompass all the addresses in each address set are then calculated. These two values are then converted to a PDU data binary bytes variable, of four bytes total length, with each value occupying two bytes. A PDU list variable is then generated with the first element consisting of the Modbus function code as an integer, and the second element being the PDU data variable. These PDU variables are then stored sequentially as elements of an overall PDU's list variable. The function then returns the PDU's list variable and Address Index list variable upon completion.

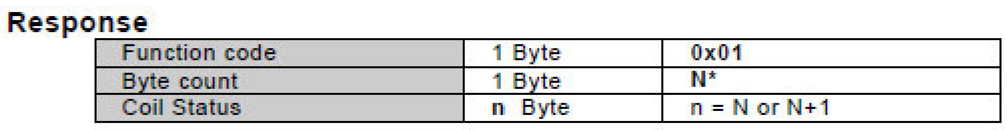

Request

Function code	1 Byte	0x01
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of coils	2 Bytes	1 to 2000 (0x7D0)

Figure 5.7 Modbus Function Code 0h01 Request Format (Modbus Organisation 2012)

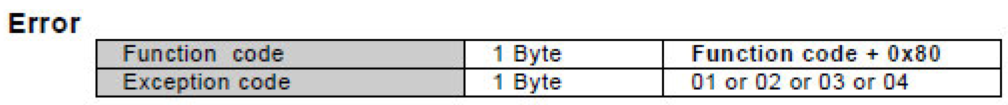
5.10.2 Gen_Read_Discrete_Output_Coil_Reply (Function Code 0h01)

This function is intended to interpret Modbus responses for 0h01 Function Code requests, formatted as shown in Figure 5.8 if the request was successful, or as shown in Figure 5.9 if unsuccessful. The function accepts a list of Modbus responses in "MBusPkt" format as its first input argument, and the associated Address Index list generated with the corresponding Modbus request PDU's as its second input argument. Each response is processed in turn as follows. First the function code of the response is checked for an error value of 0h81, a correct value of 0h01, or an invalid value. If the function code is invalid, an error is raised and code execution ended. If an error value is found, the response is passed to the "MDB_Error" function to retrieve the relevant error message. If the function code is correct, the individual bits within the response data field that correspond with the specific addresses specified within the relevant Address Index element are extracted. Once all responses have been processed a Reply Data dictionary variable is generated with an entry for each coil address. Each entry within this variable consists of a two-element list, with the first element being a Boolean value that is true if the response indicated success as opposed to error. The second element is a Boolean representation of the coil state if the response was successful, or error message text if it returned an error. The function returns the Reply Data variable upon completion.


Response

Function code	1 Byte	0x01
Byte count	1 Byte	N*
Coil Status	n Byte	n = N or N+1

Figure 5.8 Modbus Function Code 0h01 Response Format (Modbus Organisation 2012)

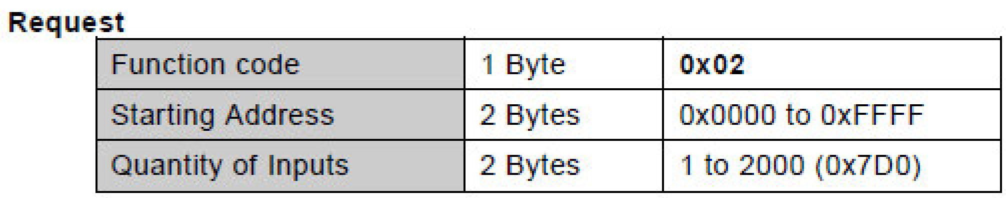

Error

Function code	1 Byte	Function code + 0x80
Exception code	1 Byte	01 or 02 or 03 or 04

Figure 5.9 Modbus Error Response Format (Modbus Organisation 2012)

5.10.3 Gen_Read_Discrete_Input_Status_PDUS (Function Code 0h02)

The purpose of this function is to generate the PDU/s for the execution of Modbus Function Code 0h02 requests, as specified in Figure 5.10. Its operation is essentially the same as "Gen_Read_Discrete_Output_Coil_PDUS", as defined in Section 5.10.1, with slight modifications for the setting of inputs.



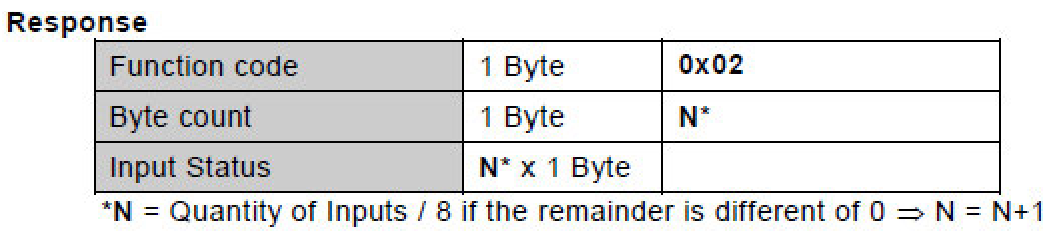
Request

Function code	1 Byte	**0x02**
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Inputs	2 Bytes	1 to 2000 (0x7D0)

Figure 5.10 Modbus Function Code 0h02 Request Format (Modbus Organisation 2012)

5.10.4 Gen_Read_Discrete_Input_Status_Reply (Function Code 0h02)

This function is intended to interpret Modbus responses for 0h02 Function Code requests, formatted as shown in Figure 5.11, if the request was successful, or as shown in Figure 5.9 if unsuccessful. Its operation is essentially the same as "Gen_Read_Discrete_Output_Coil_Reply", as defined in Section 5.10.2, with slight modifications for the setting of inputs.



Response

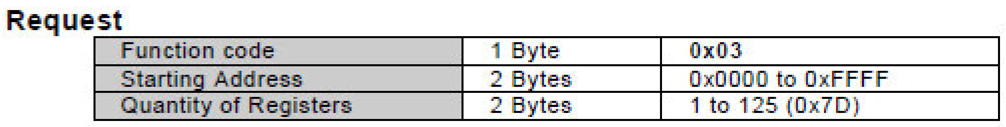
Function code	1 Byte	**0x02**
Byte count	1 Byte	**N***
Input Status	**N*** x 1 Byte	

***N** = Quantity of Inputs / 8 if the remainder is different of 0 ⇒ N = N+1

Figure 5.11 Modbus Function Code 0h02 Response Format (Modbus Organisation 2012)

5.10.5 Gen_Read_Holding_Registers_PDUS (Function Code 0h03)

The purpose of this function is to generate the PDU/s for the execution of Modbus Function Code 0h03 requests, as specified in Figure 5.12. The function accepts a list of Modbus register addresses, formatted as integers, as its sole input argument. Initially this input argument is checked to be of a list format, with an error code raised and code execution ended if it is not. A sorted list variable is then generated from the given input, in order from lowest address value to the highest. The individual addresses in the sorted list variable are then processed in order. Whilst being processed each address is checked to be of the correct integer format, and within the allowable range for this Modbus function, namely between 0h0000 and 0hFFFF. If either of these conditions is not met an error is raised and program execution is ended. The addresses are then separated into sets that meet the requirement of this Modbus function that each set does not span a range of greater than 125 subsequent register addresses. These address sets are each formatted as list variables, and are stored in order as elements of and Address Index list variable. The starting address and number of subsequent registers required to be read to encompass all the addresses in each address set are then calculated. These two values are then converted to a PDU data binary bytes variable, of four bytes total length, with each value occupying two bytes. A PDU list variable is then generated with the first element consisting of the Modbus function code as an integer, and the second element being the PDU data variable. These PDU variables are then stored sequentially as elements of an overall PDU's list variable. The function then returns the PDU's list variable and Address Index list variable upon completion.

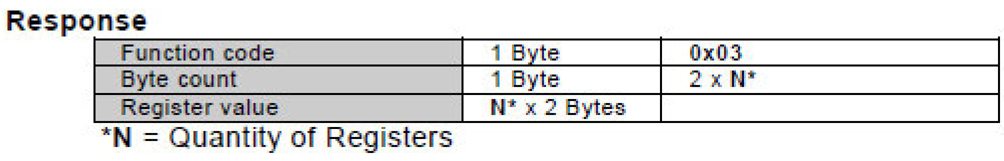

Request

Function code	1 Byte	0x03
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Registers	2 Bytes	1 to 125 (0x7D)

Figure 5.12 Modbus Function Code 0h03 Request Format (Modbus Organisation 2012)

5.10.6 Gen_Read_Holding_Registers_Reply (Function Code 0h03)

This function is intended to interpret Modbus responses for 0h03 Function Code requests, formatted as shown in Figure 5.13 if the request was successful, or as shown in Figure 5.9 if unsuccessful. The function accepts a list of Modbus responses in "MBusPkt" format as its first input argument, and the associated Address Index list generated with the corresponding Modbus request PDU's as its second input argument. A third input argument is also required, containing the configuration data variable generated by the "Read_Config" function from the device specific configuration file. Each response if processed in turn. First the function code of the response is checked for an error value of 0h83, a correct value of 0h03, or an invalid value. If the function code is invalid, an error is raised and code execution ended. If an error value is found, the response is passed to the "MDB_Error" function to retrieve the relevant error message. If the function code is correct, the individual two-byte long values within the response data field that correspond with the specific addresses specified within the relevant Address Index element are extracted. These values are then converted to either signed or unsigned integers, and scaled according to their relevant scale factor, if any, using the parameter data contained within the configuration data variable. Once all responses have been processed a Reply Data dictionary variable is generated with an entry for each register address. Each entry within this variable consists of a two-element list, with the first element being a Boolean value that is true if the response indicated success as opposed to error. The second element is the converted and scaled register value, if the response was successful, or error message text if it returned an error. The function returns the Reply Data variable upon completion.



Response

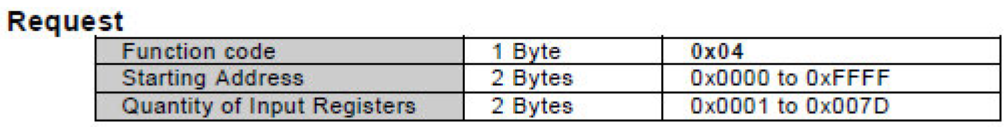
Function code	1 Byte	0x03
Byte count	1 Byte	2 x N*
Register value	N* x 2 Bytes	

*N = Quantity of Registers

Figure 5.13 Modbus Function Code 0h03 Response Format (Modbus Organisation 2012)

5.10.7 Gen_Read_Input_Registers_PDUS (Function Code 0h04)

The purpose of this function is to generate the PDU/s for the execution of Modbus Function Code 0h04 requests, as specified in Figure 5.14. Its operation is essentially the same as "Gen_Read_Holding_Registers_PDUS", as defined in Section 5.10.5, with slight modifications for the setting of input registers.



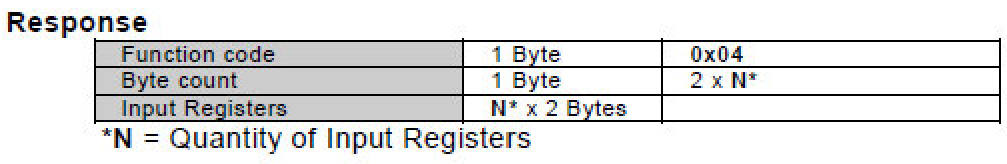
Request

Function code	1 Byte	0x04
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Input Registers	2 Bytes	0x0001 to 0x007D

Figure 5.14 Modbus Function Code 0h04 Request Format (Modbus Organisation 2012)

5.10.8 Gen_Read_Input_Registers_Reply (Function Code 0h04)

This function is intended to interpret Modbus responses for 0h04 Function Code requests, formatted as shown in Figure 5.15, if the request was successful, or as shown in Figure 5.9 if unsuccessful. Its operation is essentially the same as "Gen_Read_Holding_Registers_Reply", as defined in Section 5.10.6, with slight modifications for the setting of inputs.


Response

Function code	1 Byte	0x04
Byte count	1 Byte	2 x N*
Input Registers	N* x 2 Bytes	

*N = Quantity of Input Registers

Figure 5.15 Modbus Function Code 0h04 Response Format (Modbus Organisation 2012)

5.11 Modbus Write request/response handling

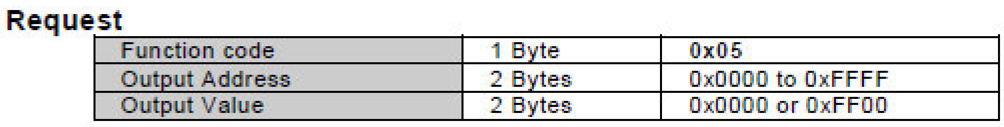
This module provides an array of functions that perform the generation of request Protocol Data Unit details, and handling of response Protocol Data Units for the following Function Codes:

- **0h05**: Force Single Coil (Bit)
- **0h06**: Preset Single Register
- **0h10**: Preset Multiple Registers

The operation of the included functions, is explained further below. This module consists of file “MDB_Write.py”, included in Appendix B.7.

5.11.1 Gen_Single_Coil_Force_PDU (Function Code 0h05)

The purpose of this function is to generate a PDU for the execution of Modbus Function Code 0h05 request, as specified in Figure 5.16. The function accepts a single of Modbus output coil address formatted as an integer as its first input argument, and a Boolean value representing the desired force state of the output coil as its second argument. Initially this input arguments are checked to be of integer and Boolean formats respectively, with an error code raised and code execution ended if it is not. The coil address and desired force state are then formatted into a PDU data binary bytes variable, of four bytes total length, with each value occupying two bytes. A value of 0h0000 and 0hFF00 are used to represent false and true force states respectively. A PDU list variable is then generated with the first element consisting of the Modbus function code as an integer, and the second element being the PDU data variable. The function then returns the PDU list variable and the output coil address upon completion.



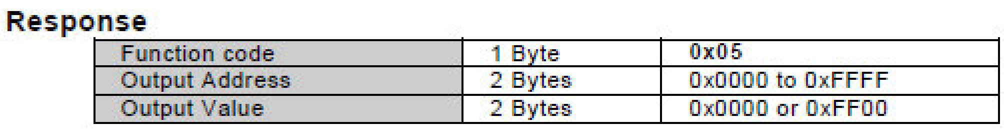
Request

Function code	1 Byte	0x05
Output Address	2 Bytes	0x0000 to 0xFFFF
Output Value	2 Bytes	0x0000 or 0xFF00

Figure 5.16 Modbus Function Code 0h05 Request Format (Modbus Organisation 2012)

5.11.2 Gen_Single_Coil_Force_Reply (Function Code 0h05)

This function is intended to interpret Modbus responses for 0h05 Function Code requests, formatted as shown in Figure 5.17 if the request was successful, or as shown in Figure 5.9 if unsuccessful. The function accepts a Modbus response in "MBusPkt" format as its first input argument, and the associated output coil address as its second input argument. First the function code of the response is checked for an error value of 0h85, a correct value of 0h05, or an invalid value. If the function code is invalid, an error is raised and code execution ended. If an error value is found, the response is passed to the "MDB_Error" function to retrieve the relevant error message. If the function code is correct, the individual two-byte long values within the response data field that correspond with the specific output coil address and output coil state are extracted. The value corresponding to the output coil address is converted to an integer and compare with the input argument output coil address. If the two addresses do not match an error is raised and code execution ended. Otherwise the output coil state field is checked for a value of 0x0000 or 0xFFFF, which correspond with false and true states respectively. A Reply Data variable it then created, which consists of a two-element list with the first element being a Boolean value that is true if the response indicated success as opposed to error. The second element is a Boolean value representing the output coils state, if the response was successful, or error message text if it returned an error. The function returns the Reply Data variable upon completion.



Response

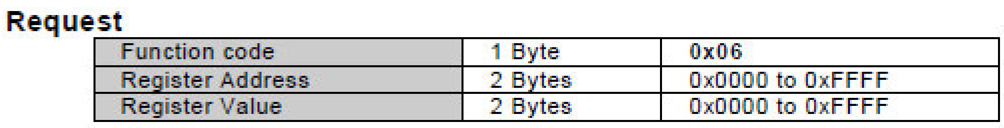
Function code	1 Byte	0x05
Output Address	2 Bytes	0x0000 to 0xFFFF
Output Value	2 Bytes	0x0000 or 0xFF00

Figure 5.17 Modbus Function Code 0h05 Response Format (Modbus Organisation 2012)

5.11.3 Gen_Preset_Single_Register_PDU (Function Code 0h06)

The purpose of this function is to generate the PDU/s for the execution of Modbus Function Code 0h06 requests, as specified in Figure 5.18. Its operation is essentially the same as

“Gen_Single_Coil_Force_PDU”, as defined in Section 5.11.1, with slight modifications for the setting of a single register as opposed to a single output coil.

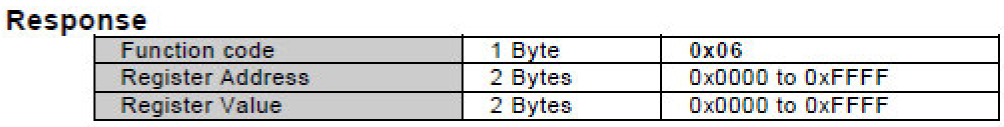

Request

Function code	1 Byte	0x06
Register Address	2 Bytes	0x0000 to 0xFFFF
Register Value	2 Bytes	0x0000 to 0xFFFF

Figure 5.18 Modbus Function Code 0h06 Request Format (Modbus Organisation 2012)

5.11.4 Gen_Preset_Single_Register_Reply (Function Code 0h06)

This function is intended to interpret Modbus responses for 0h06 Function Code requests, formatted as shown in Figure 5.19, if the request was successful, or as shown in Figure 5.9 if unsuccessful. Its operation is essentially the same as “Gen_Single_Coil_Force_Reply”, as defined in Section 5.11.2, with slight modifications for the setting of a single register as opposed to a single output coil.

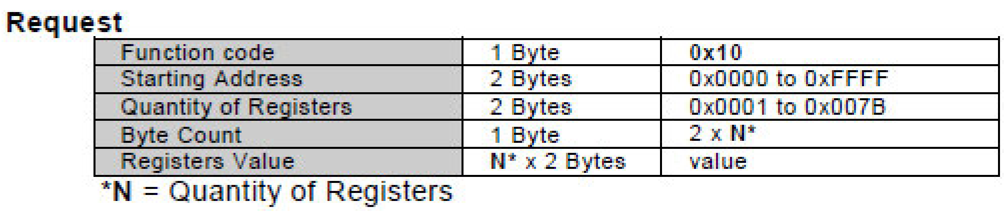

Response

Function code	1 Byte	0x06
Register Address	2 Bytes	0x0000 to 0xFFFF
Register Value	2 Bytes	0x0000 to 0xFFFF

Figure 5.19 Modbus Function Code 0h06 Response Format (Modbus Organisation 2012)

5.11.5 Gen_Preset_Multiple_Registers_PDUS (Function Code 0h10)

The purpose of this function is to generate the PDU/s for the execution of Modbus Function Code 0h10 requests, as specified in Figure 5.20. The function accepts a dictionary variable, with an entry for each Modbus register address, formatted as integers, as its sole input argument. Each entry contains the desired register value in binary bytes format. Initially this input argument is checked to be of a dictionary format, with an error code raised and code execution ended if it is not. A sorted list variable of register addresses is then generated from the given dictionary, in order from lowest address value to the highest. The individual addresses in the sorted list variable are then processed in order. Whilst being processed each address is checked to be of the correct integer format, and within the allowable range for this Modbus function, namely between 0h0000 and 0hFFFF. If either of these conditions is not met an error is raised and program execution is ended. The addresses are then separated into sets that meet the requirement of this Modbus function that each set does not span a range of greater than 123 subsequent register addresses, and additionally that all addresses in a set must be subsequent. These address sets are each formatted as list variables, and are stored in order as elements of and Address Index list variable. The corresponding values for each register address are then extracted, and checked for correct format and length. If either of these conditions are not met an error is raised and code execution is ended. The starting address and number of subsequent registers required to be written to encompass all the addresses in each address set are then calculated. These two values are then converted to a PDU data binary bytes variable, with each value occupying two bytes, followed by a single byte representing the total number of register value bytes to be written, and finished with a sequence of bytes representing the values to written for each register in order. A PDU list variable is then generated with the first element consisting of the Modbus function code as an integer, and the second element being the PDU data variable. These PDU variables are then stored sequentially as elements of an overall PDU's list variable. The function then returns the PDU's list variable and Address Index list variable upon completion.



Request

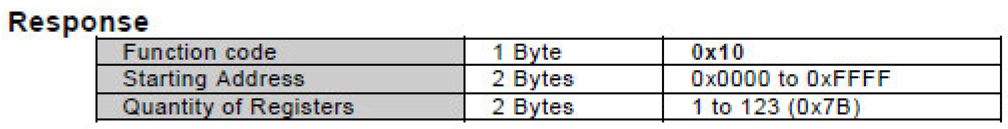
Function code	1 Byte	0x10
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Registers	2 Bytes	0x0001 to 0x007B
Byte Count	1 Byte	2 x N*
Registers Value	N* x 2 Bytes	value

*N = Quantity of Registers

Figure 5.20 Modbus Function Code 0h10 Request Format (Modbus Organisation 2012)

5.11.6 Gen_Preset_Multiple_Registers_Reply (Function Code 0h10)

This function is intended to interpret Modbus responses for 0h10 Function Code requests, formatted as shown in Figure 5.21Figure 5.13 if the request was successful, or as shown in Figure 5.9 if unsuccessful. The function accepts a list of Modbus responses in "MBusPkt" format as its first input argument, and the associated Address Index list generated with the corresponding Modbus request PDU's as its second input argument. The responses are processed in order. First the function code of the response is checked for an error value of 0h90, a correct value of 0h10, or an invalid value. If the function code is invalid, an error is raised and code execution ended. If an error value is found, the response is passed to the "MDB_Error" function to retrieve the relevant error message. If the function code is correct, the individual two-byte long value within the response data field that correspond with the starting address of the response is extracted and compared with the corresponding Address Index element starting address. If these two addresses do not match an error is raised and program execution is ended. The remaining two bytes within the data field, representing the quantity of registers written are then extracted and compared with the quantity of register addresses with the corresponding Address Index element. Once all responses have been processed a Reply Data dictionary variable is generated with an entry for each register address. Each entry within this variable consists of a two-element list, with the first element being a Boolean value that is true if the response indicated success as opposed to error. The second element is a Boolean value that is true if the response register count matched the Address Index element count, if the response was successful, or error message text if it returned an error. The function returns the Reply Data variable upon completion.



Response

Function code	1 Byte	0x10
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Registers	2 Bytes	1 to 123 (0x7B)

Figure 5.21 Modbus Function Code 0h10 Response Format (Modbus Organisation 2012)

5.12 Device configuration

This module primarily details a function that utilises functions from other relevant code modules to enact the Device Configuration operation as specified in Figure 5.1. An additional function is provided to prompt the user for the required input values to perform this operation. The exact operation of the two functions, "Modbus_Configure" and "User_Config_Input", is explained further below. This module consists of file "MDB_Configure.py", included in Appendix B.8. This module can be executed in its own right, in which case the "User_Config_Input" function is executed first, and its return values passed to the "Modbus_Configure" function as input arguments for its subsequent execution.

5.12.1 Modbus_Configure

This function requires the following input arguments:

1. Settings File Path
2. Device Configuration File Path
3. User input IP address
4. User input TCP port

The function periodically prints statement upon completion of each step in its execution to provide the user an indication of progress. First the “Read_Settings” and “Read_Config” functions are executed with their respective file paths as input arguments. The “Addr_Settings” function is then executed with the return values of the file read functions as input arguments. “Gen_Preset_Multiple_Registers_PDUS” is then executed with the address settings values given as input. The returned list of PDU’s is then passed to the “Gen_Requests” function, and the requests list returned transmitted using the “Comm” function. The “Gen_Preset_Multiple_Registers_Reply” function is then executed on the received responses, with the Address Index variable given as its second input argument. The reply data for each parameter is then printed in order for the users review.

5.12.2 User_Config_Input

This function simply prompts the user for the required input data. Firstly, the user is asked if they wish to specify a device IP address or TCP port that differ from the default values. If so, they are prompted to input these values, if not the values are set to “None” which will prompt the “Comm” function to use the default values. The function then prompts the user for the Settings File path, followed by the Device Configuration File path. All four values are then returned upon completion.

5.13 Device Interrogation

This module primarily details a function that utilises functions from other relevant code modules to enact the Device Interrogation operation as specified in Figure 5.2. An additional function is provided to prompt the user for the required input values to perform this operation. The exact operation of the two functions, “Modbus_Interrogate” and “User_Interrogate_Input”, is explained further below. This module consists of file “MDB_Interrogate.py”, included in Appendix B.9. This module can be executed in its own right, in which case the “User_Interrogate_Input” function is executed first, and its return values passed to the “Modbus_Interrogate” function as input arguments for its subsequent execution.

5.13.1 Modbus_Interrogate

This function requires the following input arguments:

1. Log File Path
2. Device Configuration File Path
3. Sampling Frequency (1-10 sec)
4. Sampling Duration (1-60 min)
5. Sampling Parameters (10 Max)
6. User input IP address
7. User input TCP port

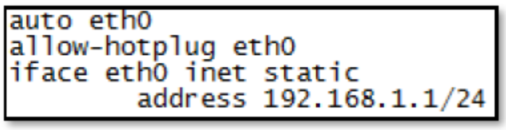
The function periodically prints statement upon completion of each step in its execution to provide the user an indication of progress. First the "Read_Config" function is executed with its respective file path as an input argument. The "Addr_Lookup" function is then executed with the return values of the file read function as its input argument. The returned lookup data is then used to determine the Modbus addresses for each of the parameters to be read, along with their read-only status. Separate address lists are then generated for read-only and non-read-only parameters. "Gen_Read_Input_Registers_PDUS", and "Gen_Read_Holding_Registers_PDUS" are then executed on these two-address list respectively. The returned lists of PDU's is then passed to the "Gen_Requests" function independently, and the requests lists returned transmitted using the "Comm" function. The "Gen_Read_Input_Registers_Reply", and "Gen_Read_Holding_Registers_Reply" functions are then executed on their respective received responses lists. The relevant Address Index variables are given as their second input arguments, the configuration data extracted by "Read_Config" given as the third. The reply data for each parameter is then printed in order for the users review, before prompting the user if they wish to commence with the sampling routine or cancel. If the user chooses to continue, the generated Modbus requests are periodically transmitted and the specified sampling rate for the specified sampling duration. Upon conclusion of the sampling duration, the collected data set is written to the specified log file.

5.13.2 User_Interrogate_Input

This function simply prompts the user for the required input data. Firstly, the user is asked if they wish to specify a device IP address or TCP port that differ from the default values. If so, they are prompted to input these values, if not the values are set to "None" which will prompt the "Comm" function to use the default values. The function then prompts the user for the Log File path, followed by the Device Configuration File path. The user is then prompted for a sampling frequency between one and ten seconds, and a sampling duration between 1 and 60 minutes. Finally, the user is prompted to enter up to ten parameter names to be sampled, and all collected input values are returned by the function upon completion.

5.14 Raspberry Pi Initial Network Configuration

In order to ensure correct operation of this software some initial configuration of the Raspberry Pi network configuration is required. Specifically the wired network interface should be configured to set the Raspberry Pi's IP address static at 192.168.1.1/24, and to allow automatic connection. This can be done by appending the text shown in Figure 5.22 to the Raspberry Pi's "/etc/network/interfaces" file (Raspberry Pi Foundation 2020). Additionally the wireless networking interface was disabled, as it was not required.



```
auto eth0
allow-hotplug eth0
iface eth0 inet static
        address 192.168.1.1/24
```

Figure 5.22 Raspberry Pi Network Configuration

6 Testing

The final Modbus TCP Configuration/Interrogation Tool was put through several stages of testing. Each individual code module was individually tested for performing its intended normal operation and error behaviours by manually generating and applying inputs that would elicit such operation and response. For example, register read and write routines were testing by successively writing all possible values of a single selected register, and confirming each write with a matching register read operation. Upon successful verification of the individual modules the two master routines were evaluated for performance. Finally, tests were performed to determine the hardware battery life duration in a variety of operating scenarios. The Configuration/Interrogation Tool was connected to the test device enclosure as shown in Figure 6.1 below. The prototype tool connected for testing is shown in Figure 6.2.

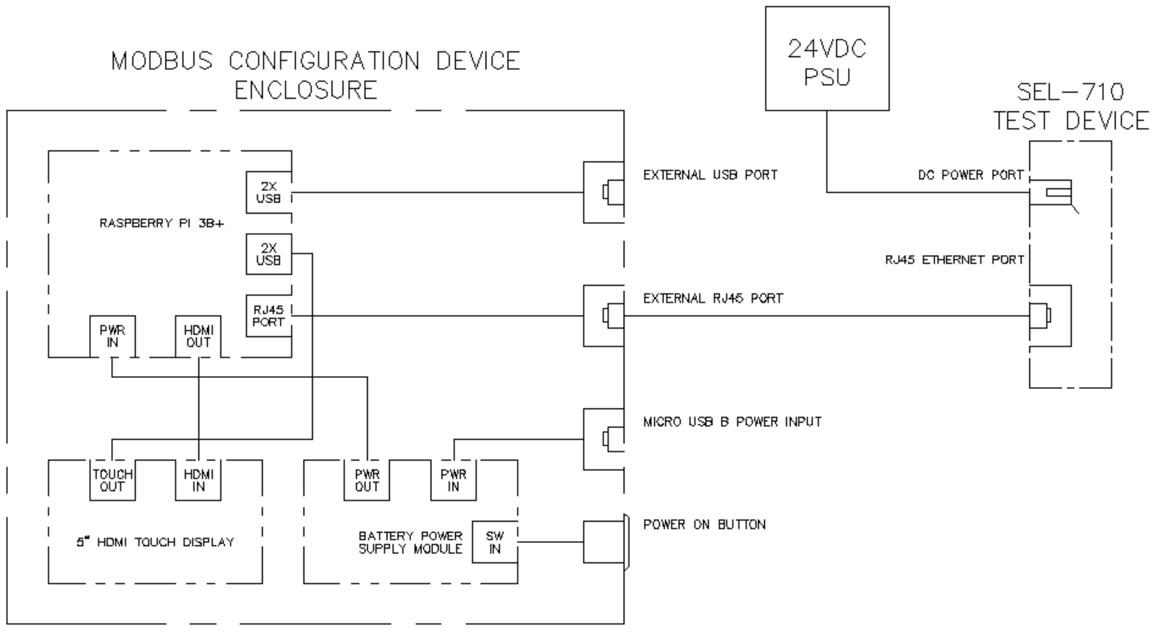
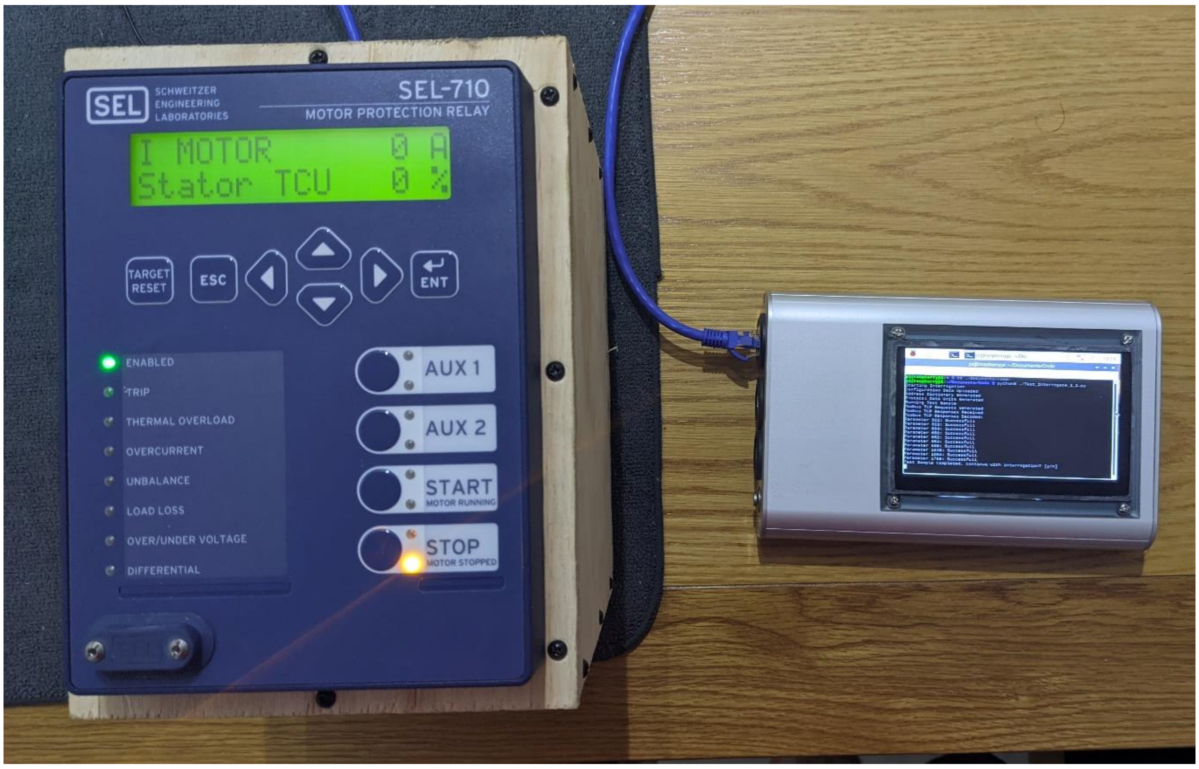


Figure 6.1 Testing Hardware Connections



```
pi@raspberrypi:~/Documents/Code
pi@raspberrypi:~ $ cd ./Documents/Code/
pi@raspberrypi:~/Documents/Code $ python3 ./Test_Interrogate_1_1.py
Starting Interrogation
Configuration Data Uploaded
Address Dictionary Generated
Protocol Data Units Generated
Running Test Sample
Modbus TCP Requests generated
Modbus TCP Responses Received
Modbus TCP Responses Decoded:
Parameter 002: Successfull
Parameter 003: Successfull
Parameter 004: Successfull
Parameter 450: Successfull
Parameter 452: Successfull
Parameter 482: Successfull
Parameter 600: Successfull
Parameter 1040: Successfull
Parameter 1006: Successfull
Parameter 1700: Successfull
Test Sample completed. continue with interrogation? [y/n]
```

Figure 6.2 Prototype tool under test

6.1 Configuration Tests

The testing of the configuration routine consisted of two phases as detailed below. A selection of written settings were manually verified on the test device after each write to confirm successful routine operation.

6.1.1 Phase One

For the first phase of testing three Settings Files were generated, with each containing all the non-read-only parameters specified in the Device Configuration File. The parameter settings in the first two files were the minimum and maximum permitted values for each parameter, respectively. The third file contained the initial parameter settings of the test device. These three files were individually input to the device configuration routine by a short python script, which also calculated the execution time of each. The results of these tests can be found in Table 6.1 below.

Test	**Duration (sec)**
Configure Default Values	13.3
Configure Minimum Values	13.6
Configure Maximum Values	13.3

Table 6.1 Configuration Routine Phase One Testing Results

As can be seen from the results the required duration did not differ appreciable between tests, as expected. It should be noted that some of the individual requests transmitted during the minimum and maximum tests returned invalid data value errors. This was determined to be due to some combinations of parameters set at the values stated for these tests violated the internal logic of the test device, and were thus rejected. It was determined upon analysis the extraction of configuration data from the Device Configuration File contributed the majority of the routine execution time.

6.1.2 Phase Two

For the second phase of testing a single settings file was generated based upon the "Nameplate" setting option as specified in the SEL-710 Installation (Schweitzer Engineering Laboratories 2015). This was done to provide a baseline "typical use" setting scenario, and the settings list of 16 parameters, provided in Appendix D, Example Typical Use Parameter Set. The settings were then applied to the device using the Configuration Routine of the developed tool, with execution timed. The settings were then entered in a scenario demonstrating the typical intended use of the Configuration Tool, including time taken to power on and connect the tool, as well as give the required user input. For comparison, the same settings were entered into the test device using the manual interface. This manual entry was performed once by a user with previous experience entering settings into a device of this type, and an unexperienced user who reviewed the relevant device documentation prior to commencing. The documentation review time prior to commencement was not included in the final completion time, but access to the documentation was allowed post commencement if required. The device settings were reset to their initial values prior to commencement of each test, with the results as shown in Table 6.2 below. Sampled settings were manually verified on the test device at several points during each test to confirm successful routine operation.

Test	**Duration (mm:ss.s)**
Manual: Inexperienced User	11:27.0
Manual: Experienced User	04:48.0
Automated: Configuration Tool	02:21.0
Automated: Routine Execution Time	00:13.5

Table 6.2 Manual Configuration vs Tool Configuration Routine Phase Two Testing Results

As can be seen from the results the Configuration Routine significantly reduces the time required to apply the desired settings to the device. The configuration tool performed the configuration function in less than half the time taken for an experienced user to perform the same task manually. It should be noted that the actual execution is consistent with those found during the Phase One tests. This shows that the Configuration Routine execution time remains consistent between setting a small quantity of parameters through to large quantities. Comparatively, manual configuration requires approximately the same time per setting, in this case approximately 18 seconds for an experienced user. Thus, as the quantity of parameters to be set increases, so too does the advantage of the Configuration Tool. For example, a not unrealistic parameter set of 32 settings could be achieved by the Configuration Tool in a quarter of the time of an experienced user.

6.2 Interrogation Tests

The Interrogation routine was subjected to a set of eight tests to determine execution duration. Four tests each were performed with sampling frequencies of once a second and once every ten seconds. The tests at each frequency were performed for the following durations:

- One minute
- Ten minutes
- Thirty minutes
- One hour

As the test device is not physically installed on an energised motor circuit during testing, for safety reasons, most of the parameter values were expected to not vary during these tests. For this reason, the following parameters were selected:

- STOPPED TIME-mm
- NUM MSG RCVD
- SET SEC
- SET MIN
- SET HOUR
- IA CURRENT
- IB CURRENT
- IC CURRENT
- IG CURRENT

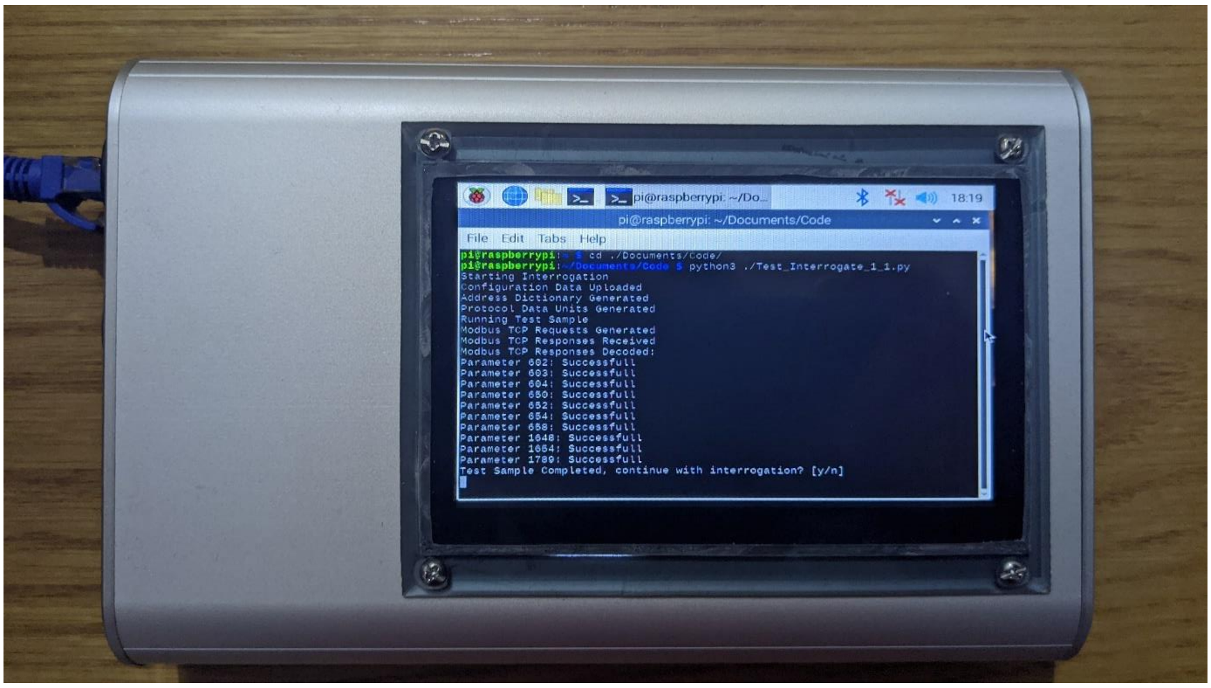
Whilst the current parameters were expected to stay at zero values for the duration of each test, the remaining parameters were based on live device time values and number of Modbus messages received, which were all expected to vary during each test. The time results of the tests are shown in Table 6.3. An excerpt of the Log file generated during the one second interval, one minute duration test is provided in Appendix E.

Test	**Duration (mm:ss)**
One per second frequency, 1-minute duration	01:15
One per second frequency, 10-minute duration	10:16
One per second frequency, 30-minute duration	30:17
One per second frequency, 60-minute duration	60:18
One per ten second frequency, 1-minute duration	01:24
One per ten second frequency, 10-minute duration	10:25
One per ten second frequency, 30-minute duration	30:26
One per ten second frequency, 60-minute duration	60:27

Table 6.3 Interrogation Routine Test Results

As can be seen from the results the time taken for execution is consistently the sum of the sampling duration, one additional sampling interval, and an additional 14 to 17 seconds. It was determined upon further analysis the vast majority of the 14 to 17 second overhead time contributed by the extraction of configuration data from the Device Configuration File. Thus, when the interrogation routine is first executed, approximately 14 to 17 seconds pass while the configuration data is loaded.

This is followed by a single sampling interval to test the configuration, and if accepted sampling continues at the sampling interval specified by the user until the sampling duration elapses. It was determined that each individual sample operation is completed in well less than 100 milliseconds, with the program waiting the remainder of the sampling interval before executing the next sample. Figure 6.3 shows the prototype tool during Interrogation testing.



```
pi@raspberrypi:~ $ cd ./Documents/Code/
pi@raspberrypi:~/Documents/Code $ python3 ./Test_Interrogate_1_1.py
Starting Interrogation
Configuration Data Uploaded
Address Dictionary Generated
Protocol Data Units Generated
Running Test Sample
Modbus TCP Requests Generated
Modbus TCP Responses Received
Modbus TCP Responses Decoded:
Parameter 602: Successfull
Parameter 603: Successfull
Parameter 604: Successfull
Parameter 650: Successfull
Parameter 652: Successfull
Parameter 654: Successfull
Parameter 658: Successfull
Parameter 1648: Successfull
Parameter 1654: Successfull
Parameter 1789: Successfull
Test Sample Completed, continue with interrogation? [y/n]
```

Figure 6.3 Prototype tool during Interrogation testing

6.3 Battery Tests

Battery tests were performed using a short python script which recorded the time elapsed since execution to two files every second. By recording the values to two files separated the script ensured that even if one file was being written when the battery completely drained, the other file would contain the uncorrupted value. This script was executed upon start-up with a fully charged battery, in three separate scenarios:

- Raspberry Pi Idle, display not powered
- Raspberry Pi Idle, display powered
- Raspberry Pi continually sampling, display powered

The results of these tests are show in Table 6.4 below.

Test	**Duration (h:mm:ss)**
Idle, No display	5:17:14
Idle, Full display	1:56:50
Active, Full Display	1:43:28

Table 6.4 Battery Life Test Results

As can be seen the device is capable of approximately two hours of continuous idle operation, or over one and a half hours of continuous active operation, without requiring recharge.

7 Conclusions and Further Work

The objective of this project was to design and develop a hardware and software solution with the goal of provided a Low-cost compact Modbus TCP based device configuration/interrogation tool. This objective has been achieved and verified by the final outcome.

A literature review of the relevant Modbus TCP theory, including technical specifications and principle of operation, was invaluable to this outcome. Combined with thorough review of the selected Modbus TCP test device, the SEL-710 Motor Protection Relay, this allowed the precise specification of the hardware and software required to perform the desired functions. Additionally, these reviews allowed the design of a suitable test assembly for the SEL-710, allowing ease of use throughout the development and testing of the Configuration/Interrogation Tool.

The developed hardware prototype has been designed to not only meet the primary objective of performing the required Modbus TCP functions, but also to meet several additional objectives. First amongst these is that the device be portable in nature. This objective has been met with the final prototype dimensions being 200 mm wide, 130 mm high, and 35 mm deep, rendering it sufficiently compact. The prototype also weighs less than 800 grams, allowing it to be easily held in one hand, and incorporates a rechargeable battery power supply for operation away from a power supply. Furthermore, the extruded aluminium enclosure and protective bezel around the touch display serve to impart sufficient mechanical strength and resilience to the prototype to allow its use in field situations. Finally, the final cost of less than $250 for the completed unit meets the stated project aim of producing a low-cost device.

The developed software modules are capable of performing the selected Modbus TCP function codes, as well as automated routines to perform the stated Configuration and Interrogation functions. The modular nature of the code produced also allows great versatility in its ability to be adapted to other non-standard tasks with minimal additional software. All software functions were found to operate correctly, including the automated Interrogation and Configuration routines. The Interrogation routine provides great benefit in its ability to perform a logging task that would not be possible to achieve via manual interface with the test device. The Configuration routine also provides significant benefit by halving the required time required to configure the test device, using a minimum number of parameters, when compared to manual configuration by an experienced operator. This benefit scales as larger numbers of parameters are required to be set, for example a set of 32 parameters could be configured by the tool in one quarter of the time taken by an experienced manual interface user, and a set of 64 parameters could be configured in one eight of the time taken by the experienced manual interface user.

In summary, the developed tool is easily hand held, of sturdy aluminium enclosed design, and of low cost. It has been tested and proven to perform the required Configuration and Interrogation functions successfully, and significantly reduce the time taken when compared to manually performing these functions. Overall, this project has answered the underlying question of the feasibility of the Modbus TCP device configuration/interrogation tool in the affirmative.

7.1 Further Work

Additional further work to complete a Graphical User Interface would greatly improve the usability of the Configuration/Interrogation Tool. This interface could also allow ease of use for not only the Configuration and Interrogation routines, but manual use of the individual Function Codes for which code functions were developed.

Development of suitable software to present the Log File results of the Interrogation routine in a variety of formats would be advantageous. The functionality to view the Log Data as a line graph for instance could be immensely useful for rapid interpretation of the acquired data.

The development of software to perform additional Modbus functions would further broaden the Tools capabilities.

8 References

Reynders, D, Mackay, S, & Wright, E 2004, *Practical Industrial Data Communications: Best Practice Techniques*, Elsevier Science & Technology, Oxford.

Park, J, Reynders, D, Mackay, S & Wright, E 2004, *Practical Industrial Data Networks: Design, Installation and Troubleshooting*, Elsevier Science & Technology, Oxford.

Modbus Organisation 2012, *Modbus Application Protocol Specification V1.1b3*, Modbus Organisation, Hopkinton, Massachusetts.

Modbus Organisation 2006, *Modbus Messaging on TCP/IP Implementation Guide V1.0b*, Modbus Organisation, Hopkinton, Massachusetts.

Modbus Organisation 2009, *Conformance Test Specification for Modbus TCP Version 3.0*, Modbus Organisation, Hopkinton, Massachusetts.

Schweitzer Engineering Laboratories 2015, *SEL-710 Motor Protection Relay Instruction Manual*, Schweitzer Engineering Laboratories, Pullman, Washington.

Raspberry Pi Foundation 2020, Raspberry Pi Foundation, Cambridge, United Kingdom, viewed 21 April 2020, <https://www.raspberrypi.org/>.

BeagleBoard.org Foundation 2019, BeagleBoard.org Foundation, Oakland Township, Michigan, viewed 22 April 2020, < https://beagleboard.org/>.

Arduino AG 2020, Arduino AG, Mainz, Rheinland-Pfalz, Germany, viewed 22 April 2020, <https://www.arduino.cc/>.

Element14 2020, Premier Farnell Limited, Leeds, England, viewed 22 April 2020, <https://au.element14.com/>.

Python 3.6 Documentation 2020, Python Software Foundation, Beaverton, Oregon, USA, viewed 30 April 2020, <https://docs.python.org/3.6/>.

Takachi Electronics Enclosure Co. Ltd. 2020, Takachi Electronics Enclosure Co. Ltd., Kawaguchi-shi, Saitama, Japan, viewed 25 June 2020, < http://www.takachi-enclosure.com/>

Appendix A, Project Specification

Project Specification

For:	Ethan Picking
Title:	Low-cost compact Modbus TCP based device configuration/interrogation tool
Major:	Electrical & Electronic Engineering
Supervisor:	Mark Phythian
Enrolment:	ENG4111 - EXT S1, 2020 ENG4112 - EXT S2, 2020
Project Aim:	To create a low-cost device capable of direct interface with Modbus TCP able devices for the purposes of configuration, commissioning, parameter verification, and fault finding.
Programme:	**Version 2, 7th March 2020**

1. Complete a literature review on Modbus TCP specifications, principle of operation, and standard methods of implementation.
2. Obtain and review relevant documentation specific to a chosen Modbus TCP capable device of significant complexity, for example a motor protection relay, at this stage intended to be a Schweitzer Engineering Laboratories model SEL-710.
3. Obtain a unit of the selected Modbus capable device and assemble with a suitable enclosure and power supply into a suitable test platform.
4. Determine standard functions that would be required to be implemented over Modbus TCP for the purpose of programming, commissioning, and fault diagnosis.
5. Determine the hardware and software requirements of a device to interface with the selected Modbus TCP device.
6. Select, obtain, and assemble suitable hardware that meets the determined hardware requirements, is resilient to physical damage, and integrates a portable power supply with provision for a simple method of recharge.
7. Develop an effective software platform capable of performing the required Modbus TCP functionality to suit the assembled hardware platform for the performance of programming, commissioning, and interrogation operations.
8. Perform functional testing of the developed device utilising the test device assembly.
9. *Optional, if time and resource permit:* Develop a suitable graphical user interface to allow simple user interface with the developed software.

Appendix B, Device Code

B.1 File Input/Output Module Code

```
'''

Module Name:  MDB_File_IO.py

Author:  Ethan Picking

Description:

This module provides functions to import data from Config and Settings files, and an export function to output log files. Config Files use YAML (.yml) format, and both Setting and Log files use CSV (.csv) format.


Copyright © Ethan Picking 2020

Use of this software is not permitted without prior consent of the author.

The author accepts no responsibility for any outcomes that may occur through the use of this software, which remain entirely at the risk of the user.

'''



#Import required yaml and csv libraries

import yaml

import csv



#Function for importing Config Files

def Read_Config(filename):

	FileDir = filename

	#Opens given filename for reading

	with open(FileDir, 'r') as ConfigFile:

		#Loads file YAML contents to variable

		ConfigData = yaml.load(ConfigFile)
	#Returns dict variable containing config data
	return ConfigData



#Function for importing Settings Files
def Read_Settings(filename):
	FileDir = filename
	#Opens given filename for reading
	with open(FileDir, 'r', newline='') as SettingFile:
		#Creates reader object for csv data
		SettingReader = csv.reader(SettingFile, dialect='excel')
		#Initialises Settings dictionary variable
		SettingData = dict()
		#Reads individual settings pairs and formats into dictionary entries
		for row in SettingReader:
			SettingData[row[0]] = float(row[1])
	#Returns dict variable containing settings data
	return SettingData



#Function for exporting Log File data
def Write_Log(filename, LogData):
	FileDir = filename
	#Opens given filename for writing, overwrites if file is preexisting
	with open(FileDir, 'w', newline='') as LogFile:
		#Creates writer object for csv data
		LogWriter = csv.writer(LogFile, dialect='excel')
		#Writes individual list entries to file
		LogWriter.writerows(LogData)
	return
```

B.2 Modbus Address Encoding/Decoding Module Code

```
'''

Module Name:  MDB_Addr.py

Author:  Ethan Picking

Description:

This module provides a function to cross reference settings names from an input settings file, and cross reference these names with specified configuration file data to provide their corresponding device specific Modbus addresses.  Data is returned as a dictionary variable matching Modbus address with correctly scaled desired settings.  An error is returned if settings are attempting to modify a read only variable, or setting value exceeds the allowable range for that setting.  An additional function is provided for generating a basic lookup dict for cross referencing of device paramater names to their respective address.


Copyright © Ethan Picking 2020

Use of this software is not permitted without prior consent of the author.

The author accepts no responsibility for any outcomes that may occur through the use of this software, which remain entirely at the risk of the user.

'''


def Addr_Settings(SettingData, ConfigData):

    #Initialise lookup dict variable

    CrossRef = dict()

    #Generate lookup dict from config data

    for Param in ConfigData:

        if ConfigData[Param]['NME'] != 'Reserved':

            CrossRef[ConfigData[Param]['NME']] = Param

    #Initialise address settings dict variable

    AddrSett = dict()

    #cycle through each parameter in the settings data

    for Param in SettingData:

        ParamAddr = CrossRef[Param]
        ParamSett = SettingData[Param]
        ParamRO = ConfigData[ParamAddr]['R/O']
        ParamType = ConfigData[ParamAddr]['TYP']
        ParamScl = ConfigData[ParamAddr]['SCL']
        ParamMin = ConfigData[ParamAddr]['MIN']
        ParamMax = ConfigData[ParamAddr]['MAX']
        #Raise error if parameter is found to be read only
        if ParamRO == True:
            print('Error: Settings File attempt to modify read-only parameter: ' + Param)
            raise SystemExit(0)
        #Raise error if Int parameter exceeds limits once scaled
        elif ParamType == 'INT':
            if ParamSett/ParamScl < ParamMin or ParamSett/ParamScl > ParamMax:
                print('Error: Parameter Setting outside limits: ' + Param)
                raise SystemExit(0)
            else:
                AddrSett[ParamAddr] = int(ParamSett/ParamScl).to_bytes(2,
byteorder = 'big', signed=True)
        #Raise error if Uint parameter exceeds limits once scaled
        elif ParamType == 'UINT':
            if ParamSett/ParamScl < ParamMin or ParamSett/ParamScl > ParamMax:
                print('Error: Parameter Setting outside limits: ' + Param)
                raise SystemExit(0)
            else:
                AddrSett[ParamAddr] = int(ParamSett/ParamScl).to_bytes(2,
byteorder = 'big')
        #Raise error if Enum or Bit Enum parameter exceeds limits
        elif ParamType == 'ENUM' or ParamType == 'BITE':
            if ParamSett < ParamMin or ParamSett > ParamMax:
                print('Error: Parameter Setting outside limits: ' + Param)
```

```
                    raise SystemExit(0)
                else:
                    AddrSett[ParamAddr] = int(ParamSett).to_bytes(2, byteorder = 'big')
        #Return address encoded paramter settings dict
        return AddrSett


def Addr_Lookup(ConfigData):
        #Initialise lookup dict variable
        CrossRef = dict()
        #Generate lookup dict from config data
        for Param in ConfigData:
                if ConfigData[Param]['NME'] != 'Reserved':
                        CrossRef[ConfigData[Param]['NME']] = [Param, ConfigData[Param]['R/O']]

        #Return name/address dict
        return CrossRef
```

B.3 Modbus Format Module Code

```
'''
Module Name:  MDB_Addr.py
Author:  Ethan Picking
Description:
This module provides a class and asssociated functions for formatting of requests into the required Modbus TCP format, and decoding of responses as required.  Generation is automated as far as possible, with errors raised if invalid data types or lengths are detected.

Copyright © Ethan Picking 2020
Use of this software is not permitted without prior consent of the author.
The author accepts no responsibility for any outcomes that may occur through the use of this software, which remain entirely at the risk of the user.
'''


#Default MBAP Header Values
MBus_Default = {'TX_ID' : 0,
                'PRTCL' : 0,
                'LNGTH' : 1,
                'UN_ID' : 0}


#Modbus ADU class, accepts bytes format response arguement as direct input, otherwise uses default values for empty packet, Length field is automatically calculated whenever a field is modified
class MBusPkt:
    def __init__(self, pkt=None):
        #If packet is given, checks for coreect format and valid length, then sets fields from given data.
        if pkt != None:
            if isinstance(pkt, bytes) == False:
                print('Error: Packet not bytes format')
                raise SystemExit(0)
            elif len(pkt) < 8:
                print('Error:  Packet too short')
                raise SystemExit(0)
            elif len(pkt[6:]) != int.from_bytes(pkt[4:6], byteorder='big'):
                print('Error:  Packet length invalid')
                raise SystemExit(0)
            else:
                self.PKT = pkt
                self.TX_ID = pkt[0:2]
                self.PRTCL = pkt[2:4]
                self.LNGTH = pkt[4:6]
                self.UN_ID = pkt[6:7]
                self.FN_CD = pkt[7:8]
                self.DATA = pkt[8:]
        #If no input packet is given, sets to defined default values
        else:
            self.TX_ID = MBus_Default['TX_ID'].to_bytes(2, byteorder='big')
            self.PRTCL = MBus_Default['PRTCL'].to_bytes(2, byteorder='big')
            self.LNGTH = MBus_Default['LNGTH'].to_bytes(2, byteorder='big')
            self.UN_ID = MBus_Default['UN_ID'].to_bytes(1, byteorder='big')
            self.FN_CD = b''
            self.DATA = b''
            self.PKT =      self.TX_ID + self.PRTCL + self.LNGTH + \
                                self.UN_ID + self.FN_CD + self.DATA

    #Defines class response value when called
    def __repr__(self):
        return self.PKT.hex()
```

```
#Defines class bytes response when called in this context
def __bytes__(self):
        return self.PKT

#Funcion for manually setting packet contents after initialisation, given as bytes
def set_PKT(self, pkt):
        #Checks for coreect format and valid length, then sets fields from given data.
        if isinstance(pkt, bytes) == False:
                print('Error: Packet not bytes format')
                raise SystemExit(0)
        elif len(pkt) < 8:
                print('Error:  Packet too short')
                raise SystemExit(0)
        elif len(pkt[6:]) != int.from_bytes(pkt[4:6], byteorder='big'):
                print('Error:  Packet length invalid')
                raise SystemExit(0)
        else:
                self.PKT = pkt
                self.TX_ID = pkt[0:2]
                self.PRTCL = pkt[2:4]
                self.LNGTH = pkt[4:6]
                self.UN_ID = pkt[6:7]
                self.FN_CD = pkt[7:8]
                self.DATA = pkt[8:]

#Fucntion for setting packet Transaction Identifier, given as an Int
def set_TX_ID(self, tx_id):
        #Checks for correct format
        if isinstance(tx_id, int) == False:
                print('Error: Transaction Identifier not int format')
```

```
            raise SystemExit(0)
        #Updates field, length, and full packet
        else:
            self.TX_ID = tx_id.to_bytes(2, byteorder = 'big')
            self.LNGTH = (2 + len(self.DATA)).to_bytes(2, byteorder='big')
            self.PKT =      self.TX_ID + self.PRTCL + self.LNGTH + \
                            self.UN_ID + self.FN_CD + self.DATA

    #Fuction for setting packet Protocol Identifier, given as an Int
    def set_PRTCL(self, prtcl):
        #Checks for correct format
        if isinstance(prtcl, int) == False:
            print('Error: Protocol Identifier not int format')
            raise SystemExit(0)
        #Updates field, length, and full packet
        else:
            self.PRTCL = prtcl.to_bytes(2, byteorder = 'big')
            self.LNGTH = (2 + len(self.DATA)).to_bytes(2, byteorder='big')
            self.PKT =      self.TX_ID + self.PRTCL + self.LNGTH + \
                            self.UN_ID + self.FN_CD + self.DATA

    #Function for setting packet Unit Identifies, given as an Int
    def set_UN_ID(self, un_id):
        #Checks for correct format
        if isinstance(un_id, int) == False:
            print('Error: Unit Identifier not int format')
            raise SystemExit(0)
        #Updates field, length, and full packet
        else:
            self.UN_ID = un_id.to_bytes(1, byteorder = 'big')
```

```
                self.LNGTH = (2 + len(self.DATA)).to_bytes(2, byteorder='big')
                self.PKT =          self.TX_ID + self.PRTCL + self.LNGTH + \
                                        self.UN_ID + self.FN_CD + self.DATA

#Function for setting packet Function Code, given as an Int
def set_FN_CD(self, fn_cd):
        #Checks for correct format
        if isinstance(fn_cd, int) == False:
                print('Error: Function Code not int format')
                raise SystemExit(0)
        #Updates field, length, and full packet
        else:
                self.FN_CD = fn_cd.to_bytes(1, byteorder = 'big')
                self.LNGTH = (2 + len(self.DATA)).to_bytes(2, byteorder='big')
                self.PKT =          self.TX_ID + self.PRTCL + self.LNGTH + \
                                        self.UN_ID + self.FN_CD + self.DATA

#Function for setting packet Data, given as Bytes
def set_DATA(self, data):
        #Checks for correct format
        if isinstance(data, bytes) == False:
                print('Error: Data not bytes format')
                raise SystemExit(0)
        #Updates field, length, and full packet
        else:
                self.DATA = data
                self.LNGTH = (2 + len(self.DATA)).to_bytes(2, byteorder='big')
                self.PKT =          self.TX_ID + self.PRTCL + self.LNGTH + \
                                        self.UN_ID + self.FN_CD + self.DATA
```

B.4 TCP Communications Module Code

```
'''
Module Name:  MDB_Comms.py
Author:  Ethan Picking
Description:
This module provides a function that uses the MDB_Format module correctly generate a list of Modbus TCP Application Data Unit Requests from a given list of Protocol Data Units.  Subsequently a function is defined for transmitting these requests via a TCP connection to the device, and formatting the received responses.


Copyright © Ethan Picking 2020
Use of this software is not permitted without prior consent of the author.
The author accepts no responsibility for any outcomes that may occur through the use of this software, which remain entirely at the risk of the user.
'''
#import required socket library for TCP communications, and select library for awaiting socket availability
import socket
import select
#import Modbus Format module
from MDB_Format import *


#Default values given for IP and Port
DEF_IP = '192.168.1.2'
DEF_PORT = 502


#Function for handling base TCP communication of given Protocol Data Units
def Comm(requests, IP=None, PORT=None):
	#Sets IP and PORT to default if not user specifid
	if IP == None:
        IP = DEF_IP
    if PORT == None:
        PORT = DEF_PORT
    #Initialise dict for respnses
    responses = dict()
    #Open TCP connection to device, using default IP and Port unless specified
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as channel:
        channel.connect((IP, PORT))
        #Send each request and receive each response
        for request in requests:
            channel.sendall(bytes(request))
            #wait for return data to receive
            Recv_Ready = select.select([channel], [], [], 1)
            if Recv_Ready[0]:
                response = channel.recv(6)
                response    +=    channel.recv(int.from_bytes(response[4:6],
byteorder='big'))
                response_TX_ID = int.from_bytes(response[0:2], byteorder='big')
                responses[response_TX_ID] = MBusPkt(response)
            else:
                channel.close()
                print('Error: Timeout waiting for device response')
                raise SystemExit(0)
        channel.close()
    #Return dict of responses
    return responses


def Gen_Requests(PDUs):
    #Initialise lists for requests
    requests = list()
```

```
responses = list()

#Initialise index for transaction identifiers

tx_index = 0

#Generation requests from given Protocol Data Units, incrementing transaction ID

for PDU in PDUs:

        request = MBusPkt()

        request.set_TX_ID(tx_index)

        tx_index += 1

        request.set_FN_CD(PDU[0])

        request.set_DATA(PDU[1])

        requests.append(request)

return requests
```

B.5 Modbus Error Handling

```
'''

Module Name:  MDB_Error.py

Author:  Ethan Picking

Description:

This module provides a dictionary and lookup function to automate Modbus error messaging.


Copyright © Ethan Picking 2020

Use of this software is not permitted without prior consent of the author.

The author accepts no responsibility for any outcomes that may occur through the use of this software, which remain entirely at the risk of the user.

'''

#Dictionary of Error responses for Modbus defined Error Codes

Error_Codes = {b'\x01': 'Error from device: Invalid Function Code',

                              b'\x02': 'Error from device: Invalid Data Addressing',

                              b'\x03': 'Error from device: Invalid Data Value',

                              b'\x04': 'Error from device: Device Failure',

                              b'\x05': 'Error from device: Acknowledge Long Duration Request',

                              b'\x06': 'Error from device: Device Busy, Please Retry',

                              b'\x08': 'Error from device: Memory Parity Error'}


#Function for extracting error code from Modbus response data

def Modbus_Error(response):

        #Extracts response error code

        Error_Code = response.DATA

        #Cross references to determine error code message

        Error_Msg = Error_Codes[Error_Code]

        #Returns error message

        return Error_Msg
```

B.6 Modbus Read request/response handling Module Code

```
'''

Module Name:  MDB_Read.py

Author:  Ethan Picking

Description:

This module provides functions for the generation of Modbus Protocol Data Units (PDU's) for the following Modbus function codes:

 - 0h01:  Read Discrete Output Coil (Bit) Status

 - 0h02:  Read Discrete Input Status

 - 0h03:  Read Holding Registers

 - 0h04:  Read Input Registers


Additional functions are provides to extract and format reply data from responses to the above function codes.


Copyright © Ethan Picking 2020

Use of this software is not permitted without prior consent of the author.

The author accepts no responsibility for any outcomes that may occur through the use of this software, which remain entirely at the risk of the user.
'''




#Import Module for Modbus Error Code Handling

from MDB_Error import *




'''
```

```
Functions for 0h01:  Read Discrete Output Coil (Bit) Status
'''
#Function for the generation of PDU's for discrete output reads
def Gen_Read_Discrete_Output_Coil_PDUS(Coils):
	#Define Function Code
	PDU_FN = 0x01

	#Check input is in correct format: list
	if isinstance(Coils, list) == False:
		print('Error: Input to Read Discrete Output must be a list')
		raise SystemExit(0)
	#Sort given list of coil addresses
	else:
		Addresses = list(Coils)
		Addresses.sort()

	#Initialise variable for indexing individual addresses as grouped into PDU's
	Address_Index = list()
	for Address in Addresses:
		#Check address is in Integer format
		if isinstance(Address, int) == False:
			print('Error: Element of Input list to Read Discrete Output must be Int')
			raise SystemExit(0)
		#Check address is within acceptable range as defined by Modbus
		elif Address < 0 or Address > 0xFFFF:
			print('Error: Invalid address for Read Discrete Output')
			raise SystemExit(0)
		#Group Addresses into the minimum number of sets, within the PDU 2000 coil limit
		elif Address == Addresses[0]:
			Address_Index.append(list())
```

```
                Address_Index[-1].append(Address)
            elif Address < Address_Index[-1][0] + 2000:
                Address_Index[-1].append(Address)
            else:
                Address_Index.append(list())
                Address_Index[-1].append(Address)

    #Initialise variable PDU's
    PDUs = list()
    #Generate PDU's according to Function Code Format
    for Element in Address_Index:
        PDU_Address = Element[0].to_bytes(2, byteorder='big')
        PDU_Length = (1 + Element[-1] - Element[0]).to_bytes(2, byteorder='big')
        PDU_Data = PDU_Address + PDU_Length
        PDUs.append([PDU_FN, PDU_Data])

    #Return list of PDU's, and Address_Index for use in decoding responses
    return [PDUs, Address_Index]




#Function for the extraction of reply data from discrete output read responses
def Gen_Read_Discrete_Output_Coil_Reply(Responses, Address_Index):
    #Define Function Code
    PDU_FN = 0x01

    #Initialise dict variable for reply data
    Reply_Data = dict()

    #Decode reply data using given Address Index
    for Index in range(0, len(Address_Index)):

        #If function code indicates error, pass to error handling module function to determine
correct error message, return data entry with false success flag, and error message text

        if int.from_bytes(Responses[Index].FN_CD, byteorder='big') == PDU_FN + 0x80:

            Error = Modbus_Error(Responses[Index])

            for Element in Address_Index[Index]:

                Reply_Data[Element] = [False, Error]

        #Extract individual bits corresponding to requested addresses, return with True
success flag and output state

        elif int.from_bytes(Responses[Index].FN_CD, byteorder='big') == PDU_FN:

            Data = int.from_bytes(Responses[Index].DATA[1:], byteorder='little')

            for Element in Address_Index[Index]:

                Mask = 1 << (Element - Address_Index[Index][0])

                if Mask & Data == Mask:

                    Reply_Data[Element] = [True, True]

                elif Mask & Data == 0:

                    Reply_Data[Element] = [True, False]

        #Error if response function code is invalid

        else:

            print('Error: Invalid response function code')

            raise SystemExit(0)


    #Return reply data dict

    return Reply_Data




'''

Functions for 0h02:  Read Discrete Input Status

'''
```

```
#Function for the generation of PDU's for discrete input reads
def Gen_Read_Discrete_Input_Status_PDUS(Inputs):
	#Define Function Code
	PDU_FN = 0x02

	#Check input is in correct format: list
	if isinstance(Inputs, list) == False:
		print('Error: Input to Read Discrete Input must be a list')
		raise SystemExit(0)
	#Sort given list of input addresses
	else:
		Addresses = list(Inputs)
		Addresses.sort()

	#Initialise variable for indexing individual addresses as grouped into PDU's
	Address_Index = list()
	for Address in Addresses:
		#Check address is in Integer format
		if isinstance(Address, int) == False:
			print('Error: Element of Input list to Read Discrete Input must be Int')
			raise SystemExit(0)
		#Check address is within acceptable range as defined by Modbus
		elif Address < 0 or Address > 0xFFFF:
			print('Error: Invalid address for Read Discrete Input')
			raise SystemExit(0)
		#Group Addresses into the minimum number of sets, within the PDU 2000 input limit
		elif Address == Addresses[0]:
			Address_Index.append(list())
			Address_Index[-1].append(Address)
		elif Address < Address_Index[-1][0] + 2000:
```

```
                Address_Index[-1].append(Address)
            else:
                Address_Index.append(list())
                Address_Index[-1].append(Address)

    #Initialise variable PDU's
    PDUs = list()
    #Generate PDU's according to Function Code Format
    for Element in Address_Index:
        PDU_Address = Element[0].to_bytes(2, byteorder='big')
        PDU_Length = (1 + Element[-1] - Element[0]).to_bytes(2, byteorder='big')
        PDU_Data = PDU_Address + PDU_Length
        PDUs.append([PDU_FN, PDU_Data])

    #Return list of PDU's, and Address_Index for use in decoding responses
    return [PDUs, Address_Index]




#Function for the extraction of reply data from discrete input read responses
def Gen_Read_Discrete_Input_Status_Reply(Responses, Address_Index):
    #Define Function Code
    PDU_FN = 0x02

    #Initialise dict variable for reply data
    Reply_Data = dict()

    #Decode reply data using given Address Index
    for Index in range(0, len(Address_Index)):
```

```
                #If function code indicates error, pass to error handling module function to determine
correct error message, return data entry with false success flag, and error message text

                if int.from_bytes(Responses[Index].FN_CD, byteorder='big') == PDU_FN + 0x80:

                    Error = Modbus_Error(Responses[Index])

                    for Element in Address_Index[Index]:

                        Reply_Data[Element] = [False, Error]

                #Extract individual bits corresponding to requested addresses, return with True
success flag and input state

                elif int.from_bytes(Responses[Index].FN_CD, byteorder='big') == PDU_FN:

                    Data = int.from_bytes(Responses[Index].DATA[1:], byteorder='little')

                    for Element in Address_Index[Index]:

                        Mask = 1 << (Element - Address_Index[Index][0])

                        if Mask & Data == Mask:

                            Reply_Data[Element] = [True, True]

                        elif Mask & Data == 0:

                            Reply_Data[Element] = [True, False]

                #Error if response function code is invalid

                else:

                    print('Error: Invalid response function code')

                    raise SystemExit(0)


            #Return reply data dict

            return Reply_Data




'''

Functions for 0h03:  Read Holding Registers

'''

#Function for the generation of PDU's for Holding Register Reads
```

```
def Gen_Read_Holding_Registers_PDUS(Registers):
        #Define Function Code
        PDU_FN = 0x03

        #Check input is in correct format: list
        if isinstance(Registers, list) == False:
                print('Error: Input to Read Holding Registers must be a list')
                raise SystemExit(0)
        #Sort given list of register addresses
        else:
                Addresses = list(Registers)
                Addresses.sort()

        #Initialise variable for indexing individual addresses as grouped into PDU's
        Address_Index = list()
        for Address in Addresses:
                #Check address is in Integer format
                if isinstance(Address, int) == False:
                        print('Error: Element of Input list to Read Holding Registers must be Int')
                        raise SystemExit(0)
                #Check address is within acceptable range as defined by Modbus
                elif Address < 0 or Address > 0xFFFF:
                        print('Error: Invalid address for Read Holding Register')
                        raise SystemExit(0)
                #Group Addresses into the minimum number of sets, within the PDU 125 register limit
                elif Address == Addresses[0]:
                        Address_Index.append(list())
                        Address_Index[-1].append(Address)
                elif Address < Address_Index[-1][0] + 125:
                        Address_Index[-1].append(Address)
```

```
		else:
			Address_Index.append(list())
			Address_Index[-1].append(Address)

	#Initialise variable PDU's
	PDUs = list()
	#Generate PDU's according to Function Code Format
	for Element in Address_Index:
		PDU_Address = Element[0].to_bytes(2, byteorder='big')
		PDU_Length = (1 + Element[-1] - Element[0]).to_bytes(2, byteorder='big')
		PDU_Data = PDU_Address + PDU_Length
		PDUs.append([PDU_FN, PDU_Data])

	#Return list of PDU's, and Address_Index for use in decoding responses
	return [PDUs, Address_Index]


#Function for the extraction of reply data from holding register read responses
def Gen_Read_Holding_Registers_Reply(Responses, Address_Index, Config_Data):
	#Define Function Code
	PDU_FN = 0x03

	#Initialise dict variable for reply data
	Reply_Data = dict()

	#Decode reply data using given Address Index
	for Index in range(0, len(Address_Index)):
		#If function code indicates error, pass to error handling module function to determine
correct error message, return data entry with false success flag, and error message text
		if int.from_bytes(Responses[Index].FN_CD, byteorder='big') == PDU_FN + 0x80:
```

```
                    Error = Modbus_Error(Responses[Index])
                    for Element in Address_Index[Index]:
                        Reply_Data[Element] = [False, Error]
                #Extract individual registers corresponding to requested addresses, return with True
success flag and register value
                elif int.from_bytes(Responses[Index].FN_CD, byteorder='big') == PDU_FN:
                    Data = Responses[Index].DATA[1:]
                    for Element in Address_Index[Index]:
                        Reg_Index = (Element - Address_Index[Index][0]) * 2
                        #Determine if value is signed or not and convert
                        if Config_Data[Element]['TYP'] == 'INT':
                            Reg_Val    =    int.from_bytes(Data[Reg_Index:Reg_Index+2],
byteorder = 'big', signed=True) * Config_Data[Element]['SCL']
                        elif Config_Data[Element]['TYP'] == 'UINT':
                            Reg_Val    =    int.from_bytes(Data[Reg_Index:Reg_Index+2],
byteorder = 'big') * Config_Data[Element]['SCL']
                        else:
                            Reg_Val    =    int.from_bytes(Data[Reg_Index:Reg_Index+2],
byteorder = 'big')
                        Reply_Data[Element] = [True, Reg_Val]
                #Error if response function code is invalid
                else:
                    print('Error: Invalid response function code')
                    raise SystemExit(0)

        #Return reply data dict
        return Reply_Data



'''
```

```
Functions for 0h04:  Read Input Registers
'''
#Function for the generation of PDU's for Input Register Reads
def Gen_Read_Input_Registers_PDUS(Registers):
	#Define Function Code
	PDU_FN = 0x04

	#Check input is in correct format: list
	if isinstance(Registers, list) == False:
		print('Error: Input to Read Input Registers must be a list')
		raise SystemExit(0)
	#Sort given list of register addresses
	else:
		Addresses = list(Registers)
		Addresses.sort()

	#Initialise variable for indexing individual addresses as grouped into PDU's
	Address_Index = list()
	for Address in Addresses:
		#Check address is in Integer format
		if isinstance(Address, int) == False:
			print('Error: Element of Input list to Read Input Registers must be Int')
			raise SystemExit(0)
		#Check address is within acceptable range as defined by Modbus
		elif Address < 0 or Address > 0xFFFF:
			print('Error: Invalid address for Read Input Register')
			raise SystemExit(0)
		#Group Addresses into the minimum number of sets, within the PDU 125 register limit
		elif Address == Addresses[0]:
			Address_Index.append(list())
```

```
                    Address_Index[-1].append(Address)
                elif Address < Address_Index[-1][0] + 125:
                    Address_Index[-1].append(Address)
                else:
                    Address_Index.append(list())
                    Address_Index[-1].append(Address)

        #Initialise variable PDU's
        PDUs = list()
        #Generate PDU's according to Function Code Format
        for Element in Address_Index:
            PDU_Address = Element[0].to_bytes(2, byteorder='big')
            PDU_Length = (1 + Element[-1] - Element[0]).to_bytes(2, byteorder='big')
            PDU_Data = PDU_Address + PDU_Length
            PDUs.append([PDU_FN, PDU_Data])

        #Return list of PDU's, and Address_Index for use in decoding responses
        return [PDUs, Address_Index]




#Function for the extraction of reply data from input register read responses
def Gen_Read_Input_Registers_Reply(Responses, Address_Index, Config_Data):
        #Define Function Code
        PDU_FN = 0x04

        #Initialise dict variable for reply data
        Reply_Data = dict()

        #Decode reply data using given Address Index
```

```
    for Index in range(0, len(Address_Index)):

        #If function code indicates error, pass to error handling module function to determine correct error message, return data entry with false success flag, and error message text

        if int.from_bytes(Responses[Index].FN_CD, byteorder='big') == PDU_FN + 0x80:

            Error = Modbus_Error(Responses[Index])

            for Element in Address_Index[Index]:

                Reply_Data[Element] = [False, Error]

        #Extract individual registers corresponding to requested addresses, return with True success flag and register value

        elif int.from_bytes(Responses[Index].FN_CD, byteorder='big') == PDU_FN:

            Data = Responses[Index].DATA[1:]

            for Element in Address_Index[Index]:

                Reg_Index = (Element - Address_Index[Index][0]) * 2

                #Determine if value is signed or not and convert, apply scaling factor

                if Config_Data[Element]['TYP'] == 'INT':

                    Reg_Val = int.from_bytes(Data[Reg_Index:Reg_Index+2], byteorder = 'big', signed=True) * Config_Data[Element]['SCL']

                elif Config_Data[Element]['TYP'] == 'UINT':

                    Reg_Val = int.from_bytes(Data[Reg_Index:Reg_Index+2], byteorder = 'big') * Config_Data[Element]['SCL']

                else:

                    Reg_Val = int.from_bytes(Data[Reg_Index:Reg_Index+2], byteorder = 'big')

                Reply_Data[Element] = [True, Reg_Val]

        #Error if response function code is invalid

        else:

            print('Error: Invalid response function code')

            raise SystemExit(0)


    #Return reply data dict

    return Reply_Data
```

B.7 Modbus Write request/response handling

'''

Module Name: MDB_Write.py

Author: Ethan Picking

Description:

This module provides functions for the generation of Modbus Protocol Data Units (PDU's) for the following Modbus function codes:

 - 0h05: Force Single Coil (Bit)

 - 0h06: Preset Single Register

 - 0h10: Preset Multiple Registers

Additional functions are provides to extract and format reply data from responses to the above function codes.

Copyright © Ethan Picking 2020

Use of this software is not permitted without prior consent of the author.

The author accepts no responsibility for any outcomes that may occur through the use of this software, which remain entirely at the risk of the user.

'''

#Import Module for Modbus Error Code Handling

from MDB_Error import *

'''

Functions for 0h05: Force Single Coil (Bit)

```
'''
#Function for the generation of PDU's for Single Coil Forces
def Gen_Single_Coil_Force_PDU(Coil, State):
        #Define Function Code
        PDU_FN = 0x05

        #Check input is in correct format: int
        if isinstance(Coil, int) == False:
                print('Error: Coil input to Single Coil Force must be int')
                raise SystemExit(0)
        #Check input is in correct format: bool
        elif isinstance(State, bool) == False:
                print('Error: State input to Single Coil Force must be bool')
                raise SystemExit(0)
        #Set Coil Address
        else:
                Address = Coil

        #Generate PDU according to Function Code Format
        PDU_Address = Address.to_bytes(2, byteorder='big')
        if State == True:
                PDU_State = 0xFF00.to_bytes(2, byteorder='big')
        else:
                PDU_State = 0x0000.to_bytes(2, byteorder='big')
        PDU_Data = PDU_Address + PDU_State
        PDU = [PDU_FN, PDU_Data]

        #Return PDU, and Address for use in decoding response
        return [PDU, Address]
```

```
#Function for the extraction of reply data from Single Coil Force responses
def Gen_Single_Coil_Force_Reply(Response, Address):
    #Define Function Code
    PDU_FN = 0x05

    #Initialise dict variable for reply data
    Reply_Data = dict()

    #If function code indicates error, pass to error handling module function to determine correct
error message, return data entry with false success flag, and error message text
    if int.from_bytes(Response.FN_CD, byteorder='big') == PDU_FN + 0x80:
        Error = Modbus_Error(Response)
        Reply_Data[Address] = [False, Error]
    elif int.from_bytes(Response.FN_CD, byteorder='big') == PDU_FN:
        Data = Response.DATA
        #If response address does not match given address, raise error
        if int.from_bytes(Data[0:2], byteorder='big') != Address:
            print('Error: Invalid response address')
            raise SystemExit(0)
        #Extract output state feedback, return with True success flag and output state
        if int.from_bytes(Data[2:], byteorder='big') == 0xFF00:
            Reply_Data[Address] = [True, True]
        elif int.from_bytes(Data[2:], byteorder='big') == 0x0000:
            Reply_Data[Address] = [True, False]
    #Error if response function code is invalid
    else:
        print('Error: Invalid response function code')
        raise SystemExit(0)
        #Return reply data dict

        return Reply_Data







'''

Functions for 0h06:  Preset Single Register

'''

#Function for the generation of a PDU for Presetting a Single Register

def Gen_Preset_Single_Register_PDU(Register, Value):

        #Define Function Code

        PDU_FN = 0x06


        #Check register is in correct format: int

        if isinstance(Register, int) == False:

                print('Error: Register input to Preset Single Register must be int')

                raise SystemExit(0)

        #Check value is in correct format: bytes

        elif isinstance(Value, bytes) == False:

                print('Error: Value input to Preset Single Register must be bytes')

                raise SystemExit(0)

        #Check value is required length

        elif len(Value) != 2:

                print('Error: Value input to Preset Single Register must be 2 bytes long')

                raise SystemExit(0)

        #Set register address

        else:

                Address = Register
```

```
	#Generate PDU according to Function Code Format
	PDU_Address = Address.to_bytes(2, byteorder='big')
	PDU_Value = Value
	PDU_Data = PDU_Address + PDU_Value
	PDU = [PDU_FN, PDU_Data]

	#Return PDU, and Address for use in decoding response
	return [PDU, Address]




#Function for the extraction of reply data from Preset Single Register responses
def Gen_Preset_Single_Register_Reply(Response, Address):
	#Define Function Code
	PDU_FN = 0x06

	#Initialise dict variable for reply data
	Reply_Data = dict()

	#If function code indicates error, pass to error handling module function to determine correct
error message, return data entry with false success flag, and error message text
	if int.from_bytes(Response.FN_CD, byteorder='big') == PDU_FN + 0x80:
		Error = Modbus_Error(Response)
		Reply_Data[Address] = [False, Error]
	elif int.from_bytes(Response.FN_CD, byteorder='big') == PDU_FN:
		Data = Response.DATA
		#If response address does not match given address, raise error
		if int.from_bytes(Data[0:2], byteorder='big') != Address:
			print('Error: Invalid response address')
```

```
                        raise SystemExit(0)
                #Extract register value feedback, return with True success flag and register value
                Reply_Data[Address] = [True, Data[2:]]
        #Error if response function code is invalid
        else:
                print('Error: Invalid response function code')
                raise SystemExit(0)

        #Return reply data dict
        return Reply_Data



'''
Functions for 0h10:  Preset Multiple Registers
'''
#Function for the generation of PDU's for Presetting Multiple Registers
def Gen_Preset_Multiple_Registers_PDUS(Registers_Values):
        #Define Function Code
        PDU_FN = 0x10

        #Check input is in correct format: dict
        if isinstance(Registers_Values, dict) == False:
                print('Error: Input to Preset Multiple Registers must be a dict')
                raise SystemExit(0)
        #Sort given list of register address/value pairs
        else:
                Addresses = list(Registers_Values.keys())
                Addresses.sort()
                Addresses_Values = dict(Registers_Values)
        #Initialise variable for indexing individual addresses as grouped into PDU's
        Address_Index = list()
        for Address in Addresses:
                #Check address is in Integer format
                if isinstance(Address, int) == False:
                        print('Error: Address keys input to Preset Multiple Registers must be Int')
                        raise SystemExit(0)
                #Check address is within acceptable range as defined by Modbus
                elif Address < 0 or Address > 0xFFFF:
                        print('Error: Invalid address for Preset Multiple Registers')
                        raise SystemExit(0)
                #Group Addresses into the minimum number of sets, within the PDU 123 register limit
                elif Address == Addresses[0]:
                        Address_Index.append(list())
                        Address_Index[-1].append(Address)
                #ensure withing 123 register limit, and consecutive registers
                elif (Address < Address_Index[-1][0] + 123) and (Address - Address_Index[-1][-1] ==
1):
                        Address_Index[-1].append(Address)
                else:
                        Address_Index.append(list())
                        Address_Index[-1].append(Address)

        #Initialise variable PDU's
        PDUs = list()
        #Generate PDU's according to Function Code Format
        for Element in Address_Index:
                Reg_Quantity = (1 + Element[-1] - Element[0])
                PDU_Address = Element[0].to_bytes(2, byteorder='big')
```

```
			PDU_Length = Reg_Quantity.to_bytes(2, byteorder='big')
			PDU_Count = (Reg_Quantity * 2).to_bytes(1, byteorder='big')
			PDU_Values = bytes()
			for Address in Element:
				Value = Addresses_Values[Address]
				#Check value is in Bytes format
				if isinstance(Value, bytes) == False:
					print('Error: Values input to Preset Multiple Registers must be in Bytes')
					raise SystemExit(0)
				#Check Value is correct length
				elif len(Value) != 2:
					print('Error: Error: Value input to Preset Multiple Registers must be 2 bytes long')
					raise SystemExit(0)
				else:
					PDU_Values += Value
			PDU_Data = PDU_Address + PDU_Length + PDU_Count + PDU_Values
			PDUs.append([PDU_FN, PDU_Data])

	#Return list of PDU's, and Address_Index for use in decoding responses
	return [PDUs, Address_Index]




#Function for the extraction of reply data from Preset Multiple Registers responses
def Gen_Preset_Multiple_Registers_Reply(Responses, Address_Index):
	#Define Function Code
	PDU_FN = 0x10
```

```
    #Initialise dict variable for reply data
    Reply_Data = dict()

    #Decode reply data using given Address Index
    for Index in range(0, len(Address_Index)):
        #If function code indicates error, pass to error handling module function to determine correct error message, return data entry with false success flag, and error message text
        if int.from_bytes(Responses[Index].FN_CD, byteorder='big') == PDU_FN + 0x80:
            Error = Modbus_Error(Responses[Index])
            for Element in Address_Index[Index]:
                Reply_Data[Element] = [False, Error]
        #Check correct format of response data, return with True success flag and reply correct format flag
        elif int.from_bytes(Responses[Index].FN_CD, byteorder='big') == PDU_FN:
            Data_Address = int.from_bytes(Responses[Index].DATA[0:2], byteorder='big')
            Data_Length = int.from_bytes(Responses[Index].DATA[2:], byteorder='big')
            if Data_Address == Address_Index[Index][0] and Data_Length == (1 + Address_Index[Index][-1] - Address_Index[Index][0]):
                for Element in Address_Index[Index]:
                    Reply_Data[Element] = [True, True]
            else:
                for Element in Address_Index[Index]:
                    Reply_Data[Element] = [True, False]
        #Error if response function code is invalid
        else:
            print('Error: Invalid response function code')
            raise SystemExit(0)

    #Return reply data dict
    return Reply_Data
```

B.8 Device configuration

```
'''
Module Name:  MDB_Configure.py
Author:  Ethan Picking
Description:
This module provides a function for the automated configuring of a Modbus TCP device from user defined settings and configuration files.

Copyright © Ethan Picking 2020
Use of this software is not permitted without prior consent of the author.
The author accepts no responsibility for any outcomes that may occur through the use of this software, which remain entirely at the risk of the user.
'''



#import required modules
from MDB_File_IO import *
from MDB_Addr import *
from MDB_Write import *
from MDB_Comms import *



#Function for obtaining user inputs for IP, Port, Settings file, and Config File
def User_Config_Input():
	print('Type "y" to use different values than default IP address: 192.168.1.2 and Port: 502')
	if input() == 'y':
		print('Please enter device IP address (default: 192.168.1.2):')
		User_IP = input()
```

```
        print('Please enter device Port number (default: 502):')
        User_Port = input()
    else:
        User_IP = None
        User_Port = None
    print('Enter Settings file directory:')
    File_Sett = input()
    print('Enter Device Config file directory:')
    File_Config = input()
    return [File_Sett, File_Config, User_IP, User_Port]




#Function for configuring Modbus device
def Modbus_Configure(File_Sett, File_Config, User_IP, User_Port):
    print('Starting Configuration')

    #Uploads settings file
    Data_Sett = Read_Settings(File_Sett)
    print('Settings Uploaded')

    #Uploads device config file
    Data_Config = Read_Config(File_Config)
    print('Configuration Data Uploaded')

    #Generates settings list with modbus addresses
    Addr_Sett = Addr_Settings(Data_Sett, Data_Config)
    print('Settings Addresses Generated')

    #Generates PDU's
```

```
	[PDUs, Address_Index] = Gen_Preset_Multiple_Registers_PDUS(Addr_Sett)
	print('Protocol Data Units Generated')

	#Generated Modbus Requests
	Requests = Gen_Requests(PDUs)
	print('Modbus TCP Requests Generated')

	#Transmits requests, receives responses
	Responses = Comm(Requests, IP=User_IP, PORT=User_Port)
	print('Modbus TCP Responses Received')

	#Decodes Responses
	Reply = Gen_Preset_Multiple_Registers_Reply(Responses, Address_Index)
	print('Modbus TCP Responses Decoded:')

	#Displays operation status
	for Param in Reply:
		if Reply[Param][0] == True:
			print('Parameter ' + str(Param) + ': Successfull')
		else:
			print('Parameter ' + str(Param) + ': ' + Reply[Param][1])

	print('Configuration Concluded')
	return


#Exectues functions only if module is run, instead of imported
if __name__ == '__main__':
	[File_Sett, File_Config, User_IP, User_Port] = User_Config_Input()
	Modbus_Configure(File_Sett, File_Config, User_IP, User_Port)
```

B.9 Device Interrogation

```
'''

Module Name:  MDB_Interrogate.py

Author:  Ethan Picking

Description:

This module provides a function for the automated interrogation of a Modbus TCP device, given a set of parameters, a sampling frequency, and sampling duration.  Logged data is exported to a specified log file.


Copyright © Ethan Picking 2020

Use of this software is not permitted without prior consent of the author.

The author accepts no responsibility for any outcomes that may occur through the use of this software, which remain entirely at the risk of the user.

'''


#import time module for scheduling

import time


#import required modules

from MDB_File_IO import *

from MDB_Addr import *

from MDB_Read import *

from MDB_Comms import *




#Function for obtaining user inputs for IP, Port, Log file, Sampling Frequency, Sampling duration, and parameters to be sampled

def User_Interrogate_Input():

	print('Type "y" to use different values than default IP address: 192.168.1.2 and Port: 502')
```

```
if input() == 'y':
    print('Please enter device IP address (default: 192.168.1.2):')
    User_IP = input()
    print('Please enter device Port number (default: 502):')
    User_Port = input()
else:
    User_IP = None
    User_Port = None
print('Enter Log file directory:')
File_Log = input()
print('Enter Device Config file directory:')
File_Config = input()
print('Enter Sampling Frequency between 1 and 10 seconds:')
Samp_Freq = int(input())
if Samp_Freq < 0 or Samp_Freq > 10:
    print('Error: Sample Frequency out of range')
    raise SystemExit(0)
print('Enter Sampling Duration between 1 and 60 minutes:')
Samp_Dur = int(input())
if Samp_Freq < 0 or Samp_Freq > 60:
    print('Error: Sample Duration out of range')
    raise SystemExit(0)
print('Enter up to 10 parameters to sample, press Enter after each:')
count = 0
Samp_Params = list()
while count < 10:
    Param = input()
    if Param == '':
        break
    count += 1
```

```
        Samp_Params.append(Param)
    return [File_Log, File_Config, Samp_Freq, Samp_Dur, Samp_Params, User_IP, User_Port]



#Function for interrogating Modbus device
def Modbus_Interrogate(File_Log, File_Config, Samp_Freq, Samp_Dur, Samp_Params, User_IP, User_Port):
    print('Starting Interrogation')
    #Uploads device config file
    Data_Config = Read_Config(File_Config)
    print('Configuration Data Uploaded')

    #Generates address lookup dict
    Addr_Sett = Addr_Lookup(Data_Config)
    print('Address Dictionary Generated')

    #Generates address lists
    Addr_Input = list()
    Addr_Hold = list()
    for Param in Samp_Params:
        if Addr_Sett[Param][1] == True:
            Addr_Input.append(Addr_Sett[Param][0])
        else:
            Addr_Hold.append(Addr_Sett[Param][0])

    #Generates Holding regist PDU's
    [PDUs_Hold, Address_Index_Hold] = Gen_Read_Holding_Registers_PDUS(Addr_Hold)
    [PDUs_Input, Address_Index_Input] = Gen_Read_Input_Registers_PDUS(Addr_Input)
    print('Protocol Data Units Generated')
```

```
	print('Running Test Sample')


	#Generated Modbus Requests
	Requests_Hold = Gen_Requests(PDUs_Hold)
	Requests_Input = Gen_Requests(PDUs_Input)
	print('Modbus TCP Requests Generated')


	#Transmits requests, receives responses
	if len(Requests_Hold) > 0:
		Responses_Hold = Comm(Requests_Hold, IP=User_IP, PORT=User_Port)
	else:
		Responses_Hold = []
	if len(Requests_Input) > 0:
		Responses_Input = Comm(Requests_Input, IP=User_IP, PORT=User_Port)
	else:
		Responses_Input = []
	print('Modbus TCP Responses Received')


	#Decodes Responses
	Reply_Hold = Gen_Read_Holding_Registers_Reply(Responses_Hold, Address_Index_Hold,
Data_Config)
	Reply_Input = Gen_Read_Input_Registers_Reply(Responses_Input, Address_Index_Input,
Data_Config)
	Reply = dict()
	Reply.update(Reply_Hold)
	Reply.update(Reply_Input)
	print('Modbus TCP Responses Decoded:')


	#Displays operation status
```

```
for Param in Reply:
        if Reply[Param][0] == True:
                print('Parameter ' + str(Param) + ': Successfull')
        else:
                print('Parameter ' + str(Param) + ': ' + Reply[Param][1])

#Queries user to determine where to continue with interrogation post test
print('Test Sample Completed, continue with interrogation? [y/n]')
Continue_Flag = False
while True:
        Entry = input()
        if Entry == 'y':
                Continue_Flag = True
                break
        elif Entry == 'n':
                Continue_Flag = False
                break
        else:
                print('Invalid entry.  Enter "y" or "n":')

#Interrogates if user selects continue
if Continue_Flag == True:
        print('Proceeding with interrogation...')
        #initialise data log variable
        Data_Log = list()

        #Create data log heading values
        Data_Log.append(['TIME(s)'])
        Reply_keys = list(Reply.keys())
        Reply_keys.sort()
```

```
for Param in Reply_keys:
    Data_Log[0].append(Data_Config[Param]['NME'])


#Set sampling parameters
Samp_Max = Samp_Dur * 60 / Samp_Freq
Samp_Count = 0


#perform sampling at desired frequency, for desired duration
while Samp_Count <= Samp_Max:
    Time_Samp = time.time()
    if len(Requests_Hold) > 0:
        Responses_Hold = Comm(Requests_Hold, IP=User_IP, PORT=User_Port)
    else:
        Responses_Hold = []
    if len(Requests_Input) > 0:
        Responses_Input = Comm(Requests_Input, IP=User_IP, PORT=User_Port)
    else:
        Responses_Input = []
    Reply_Hold = Gen_Read_Holding_Registers_Reply(Responses_Hold, Address_Index_Hold, Data_Config)
    Reply_Input = Gen_Read_Input_Registers_Reply(Responses_Input, Address_Index_Input, Data_Config)
    Reply = dict()
    Reply.update(Reply_Hold)
    Reply.update(Reply_Input)
    Data_Log.append([Samp_Count * Samp_Freq])
    for Param in Reply_keys:
        if Reply[Param][0] == True:
            Data_Log[-1].append(Reply[Param][1])
```

```
                else:
                    Data_Log[-1].append('ERROR')
            print('Sample ' + str(Samp_Count) + ' Complete, ' + str(int((Samp_Max-
Samp_Count)*Samp_Freq)) + ' Seconds remaining...')
            Samp_Count += 1
            while time.time() < Time_Samp + Samp_Freq:
                time.sleep(0.001)


        #write logged data to file
        print('Writing log file')
        Write_Log(File_Log, Data_Log)


    print('Interrogation Concluded')
    return




#Exectues functions only if module is run, instead of imported
if __name__ == '__main__':
    [File_Log, File_Config, Samp_Freq, Samp_Dur, Samp_Params, User_IP, User_Port] =
User_Interrogate_Input()
    Modbus_Interrogate(File_Log, File_Config, Samp_Freq, Samp_Dur, Samp_Params, User_IP,
User_Port)
```

Appendix C, Risk Assessment

The following risk assessment considers risks anticipated to be present both in the design and construction of the device, as well as its eventual use during testing. In order to assess the relative levels of risk a matrix that takes into account both consequence magnitude and likelihood has been used, as shown in Table C.1.

		Consequence			
		Minor	Moderate	Major	Catastrophic
Likelihood	Likely	Moderate	High	High	High
	Moderate	Low	Moderate	High	High
	Unlikely	Low	Low	Moderate	High
	Rare	Low	Low	Low	Moderate

Table C.1 Consequence vs. Likelihood Risk Matrix

Hazard	Risk	Controls	Risk with Controls
Mechanical injury during construction, e.g. cuts from sharp edges, injury from tool use etc.	High (Moderate, Likely)	• Utilise correct tools for the task at hand. • Use correct PPE where required, for example cut proof gloves if dealing with sharp objects • Follow manufacturer's instructions for tools use • Remove potential hazards from the work area as soon as they are formed	Low (Moderate, Unlikely)
Electric shock during construction, testing, and use	High (Catastrophic, Moderate)	• Select extra low voltage equipment to remove shock potential	Low (Minor, Unlikely)
Equipment damage due to spurious operation during prototyping	High (Major, Moderate)	• Assemble SEL-710 relay into separate standalone test unit to remove consequence of errant operation	Low (Minor, Moderate)
Strain from significant duration of physical and desk work	Moderate (Minor, Likely)	• Maintain correct postures • Break regularly to avoid strain	Low (Minor, Unlikely)

Table C.2 Project Risk Assessment

Appendix D, Example Typical Use Parameter Set

The following settings were used as an example "Typical Use" parameter set during Phase Two testing of the Configuration routine. This was intended to represent the bare minimum quantity of parameters required to be set for the test device before operation.

Parameter	**Setting**
APPLICATION	1
PHASE ROTATION	0
RATED FREQ.	0
DATE FORMAT	2
PHASE CT RATIO	100
MOTOR FLA	75
NEUTRAL CT RATIO	80
PHASE PT RATIO	35
SERVICE FACTOR	1.05
MOTOR LRA	6
LOCKD ROTOR TIME	10
NEUT OC TRIP LVL	10
NEU OC TRIP DLAY	0.5
RES OC TRIP LVL	0.5
RES OC TRIP DLAY	0.5
OUT103 FAIL-SAFE	1

Table D.1 Typical Use Parameter Set

Appendix E, Example Log File Excerpt

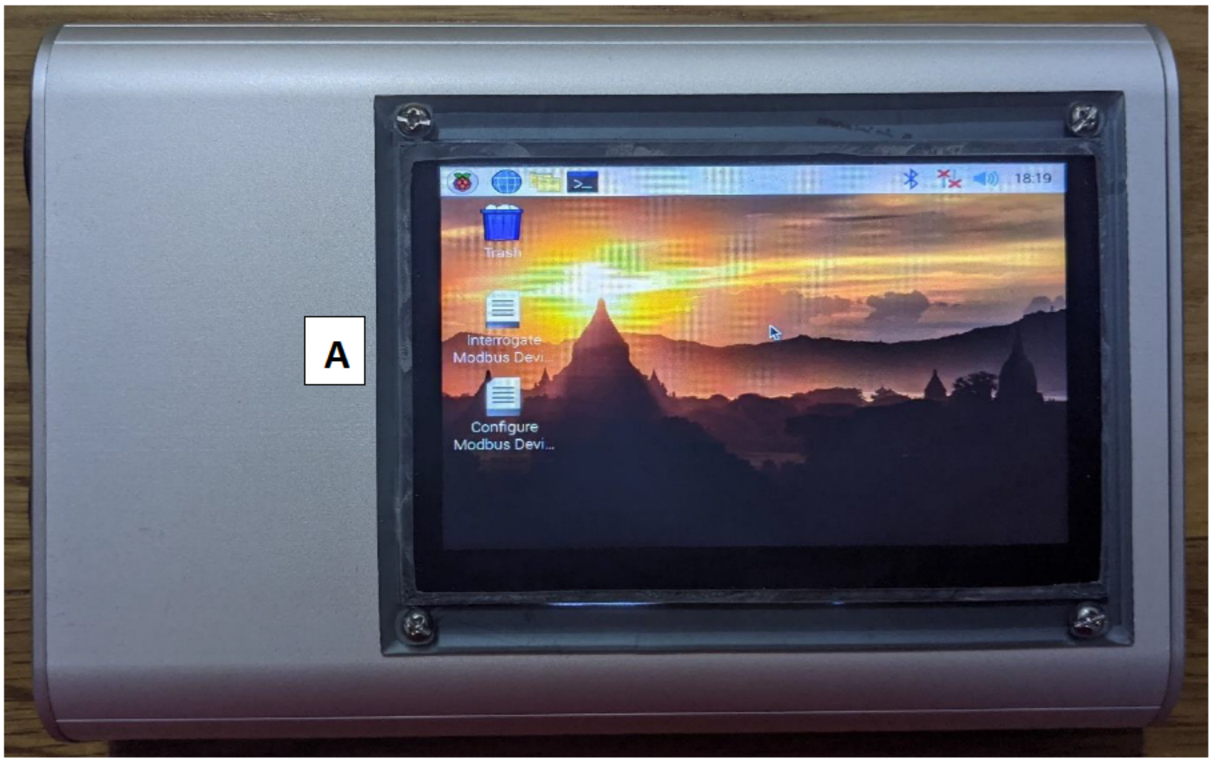
The following is an excerpt of a log file output by the Interrogation Routine, operating at one second sampling intervals.

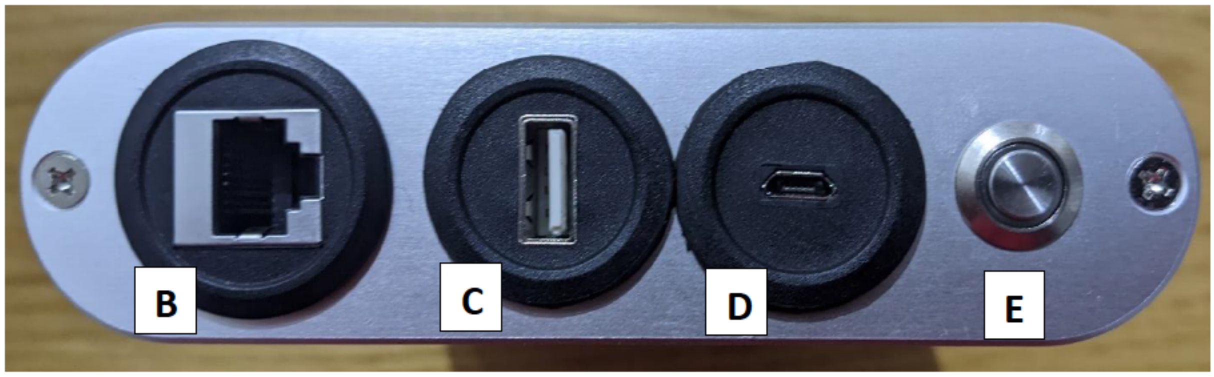
TIME (s)	SET SEC	SET MIN	SET HOUR	IA CURR ENT	IB CURR ENT	IC CURR ENT	IG CURR ENT	ELAPSED TIME-mm	STOPPED TIME-mm	NUM MSG RCVD
0	44	2	0	0	0	0	0	11	11	76
1	45	2	0	0	0	0	0	11	11	80
2	46	2	0	0	0	0	0	12	12	84
3	47	2	0	0	0	0	0	12	12	88
4	48	2	0	0	0	0	0	12	12	92
5	49	2	0	0	0	0	0	12	12	96
6	50	2	0	0	0	0	0	12	12	100
7	51	2	0	0	0	0	0	12	12	104
8	52	2	0	0	0	0	0	12	12	108
9	53	2	0	0	0	0	0	12	12	112
10	54	2	0	0	0	0	0	12	12	116
11	55	2	0	0	0	0	0	12	12	120
12	56	2	0	0	0	0	0	12	12	124
13	57	2	0	0	0	0	0	12	12	128
14	58	2	0	0	0	0	0	12	12	132
15	59	2	0	0	0	0	0	12	12	136
16	0	3	0	0	0	0	0	12	12	140
17	1	3	0	0	0	0	0	12	12	144
18	2	3	0	0	0	0	0	12	12	148
19	3	3	0	0	0	0	0	12	12	152
20	4	3	0	0	0	0	0	12	12	156
21	5	3	0	0	0	0	0	12	12	160
22	6	3	0	0	0	0	0	12	12	164
23	7	3	0	0	0	0	0	12	12	168
24	8	3	0	0	0	0	0	12	12	172
25	9	3	0	0	0	0	0	12	12	176
26	10	3	0	0	0	0	0	12	12	180
27	11	3	0	0	0	0	0	12	12	184
28	12	3	0	0	0	0	0	12	12	188
29	13	3	0	0	0	0	0	12	12	192
30	14	3	0	0	0	0	0	12	12	196

Table E.1 Interrogation Routine Log File Excerpt

Appendix F, User Instructions

Tool Components:





A. Screen
B. RJ-45 Port
C. USB Port
D. Charging Port
E. Power Button

Warning

When connecting/disconnecting to a Modbus TCP device, ensure no hazardous voltages are present on adjacent terminals and if so, isolate before proceeding.

Power On

Depress the power button momentarily, wait while device starts up until desktop is displayed as shown above.

Transferring Files

Relevant Settings, Configuration, and Log Files may be transferred to and from the device using a USB storage device connected to the external USB port, via the File Browser. Alternatively when using the Configuration or Interrogation routines, files paths may be entered when prompted that specify files located on a connected USB storage device.

Configure Modbus Device

Connect Tool to Modbus TCP device via the external RJ-45 port and an ethernet patch lead. Ensure device is set to IP address 192.168.1.2 (unless specifying your own IP address), and is ready for Modbus TCP communications. Select the “Configure Modbus Device” shortcut from the desktop, select “Execute” when prompted. When prompted complete the required user input fields. Configuration routine will now run, reporting status and completion notices. Once complete, press the enter key to close the routine. Disconnect from the Modbus TCP device.

Interrogate Modbus Device

Connect Tool to Modbus TCP device via the external RJ-45 port and an ethernet patch lead. Ensure device is set to IP address 192.168.1.2 (unless specifying your own IP address), and is ready for Modbus TCP communications. Select the “Interrogate Modbus Device” shortcut from the desktop, select “Execute” when prompted. When prompted complete the required user input fields. Interrogation routine will now run, reporting status and completion notices. Once complete, press the enter key to close the routine. Disconnect from the Modbus TCP device.

Power Off

Select Logout from the Raspberry Pi menu, choose Shutdown when prompted. Once Shutdown in complete, double press the power button to turn off the internal power supply.