BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF CONTROL AND INSTRUMENTATION

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

THE LIBRARY FOR MODBUS RTU IN FORTH LANGUAGE

KNIHOVNA PRO MODBUS RTU V JAZYCE FORTH

BACHELOR'S THESIS

BAKALÁŘSKÁ PRÁCE

AUTHOR Jakub Kouřil

AUTOR PRÁCE

SUPERVISOR prof. Ing. Pavel Jura, CSc.

VEDOUCÍ PRÁCE

BRNO 2022

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Bachelor's Thesis

Bachelor's study program **Automation and Measurement**

Department of Control and Instrumentation

Student: Jakub Kouřil ***ID:*** 220992
Year of study: 3 ***Academic year:*** 2021/22

TITLE OF THESIS:

The library for MODBUS RTU in FORTH language

INSTRUCTION:

FORTH is the minimalist programming language, the operating system and the development kit designed by Ch. Moore. FORTH supported multi-tasking and multi-users also in time when computers was single-task capable. This system could work with minimal system resources so it can run in small MCUs.
MODBUS RTU is well known open protocol mainly used in industry.
The implementation of library for working with MODBUS RTU slave device in FORTH language is the main goal of this bachelor thesis. The functionality of this library will be demonstrated on few units.
1. Study of FORTH language.
2. Make research of FORTH variants designed for MCUs and compare theirs properties.
3. Study of MODBUS RTU.
4. Implement the library for MODBUS RTU slave device in FORTH language.
5. With usage of library make few slave nodes on MODBUS RTU bus.

RECOMMENDED LITERATURE:

[1] BRODIE, Leo. Starting FORTH: an introduction to the FORTH language and operating system for beginners and professionals [online]. Englewood Cliffs, N.J.: Prentice-Hall, c1981 [cit. 2019-09-13]. ISBN 01-384-2930-8. Dostupné z: https://www.forth.com/wp-content/uploads/2018/01/Starting-FORTH.pdf

Date of project specification: 7.2.2022 ***Deadline for submission:*** 23.5.2022

Supervisor: prof. Ing. Pavel Jura, CSc.
Consultant: Ing. Vilém Kárský

doc. Ing. Václav Jirsík, CSc.
Chair of study program board

WARNING:

The author of the Bachelor's Thesis claims that by creating this thesis he/she did not infringe the rights of third persons and the personal and/or property rights of third persons were not subjected to derogatory treatment. The author is fully aware of the legal consequences of an infringement of provisions as per Section 11 and following of Act No 121/2000 Coll. on copyright and rights related to copyright and on amendments to some other laws (the Copyright Act) in the wording of subsequent directives including the possible criminal consequences as resulting from provisions of Part 2, Chapter VI, Article 4 of Criminal Code 40/2009 Coll.

ABSTRACT

The goal of this thesis is to develop a library for Modbus RTU communication using the Forth programming language. A demonstration unit is also to be created. The resulting library can be used for example in hobby projects in the field of data acquisition and automation. The thesis also contains chapters dedicated to the Modbus protocol and the Forth language itself, including descriptions of several Forth implementations for various microcontrollers. The library was developed according to the assignment using FlashForth, a Forth implementation for the Atmega and PIC microcontrollers. The demonstration unit consisting of several Arduino NANO boards was constructed, proving the library functionality.

KEYWORDS

Modbus, Forth language, library, Atmega328p, Arduino

ABSTRAKT

Cílem této práce je vyvinout knihovnu pro Modbus RTU komunikaci v programovacím jazyce Forth a vytvořit demonstrační jednotku pro ověření funkčnosti vytvořené knihovny. Tato knihovna může být využita například v projektech převážně hobby povahy v oblasti sběru dat a automatizace. Práce také obsahuje kapitoly věnující se Modbus protokolu a jazyku Forth samotnému včetně popisu několika implementací jazyka Forth pro různé mikrokontroléry. Knihovna byla dle zadání úspěšně vyvinuta za použití systému FlashForth, implementace Forth jazyka pro mikrokontroléry Atmega a PIC. Byla taktéž sestavena demonstrační jednotka skládající se z několika Arduino NANO desek pro prokázání funkčnosti knihovny.

KLÍČOVÁ SLOVA

Modbus, jazyk Forth, knihovna, Atmega328p, Arduino

Typeset by the `thesis` package, version 4.07; http://latex.feec.vutbr.cz

ROZŠÍŘENÝ ABSTRAKT

Tato práce má za cíl vytvořit knihovnu slov v programovacím jazyce Forth, která bude implementovat komunikaci pomocí protokolu Modbus RTU pro server zařízení. Tato knihovna následně může být použita pro vytváření server jednotek pro účely automatizace a sběru dat v malých projektech či hobby aplikacích. Za tímto účelem má práce také popsat protokol Modbus, samotný programovací jazyk Forth a také jeho implementace pro různé mikrokontroléry. Pro demonstraci fungování hotové knihovny je také dalším úkolem zhotovit demonstrační jednotku obsahující několik Modbus server jednotek na sběrnici.

Forth je programovací jazyk, Forth systémy však mají vlastnosti operačních systémů a vývojových prostředí. Jedná se o interpretovaný jazyk zaměřený na operaci na zásobnících. Programování nových funkcí (slov) je realizováno skrze komunikaci se systémem přes terminál. Tento jazyk a systém je používaný převážně pro embedded zařízení vzhledem k jeho malé velikosti a také k rychlosti vykonávání kódu kterou nabízí.

Modbus RTU je jednoduchý komunikační protokol modelu server/klient. Tento protokol je založený na využití funkčních kódů pro specifikaci požadavku od klienta pro server a umožnuje především jednoduché čtení dat ze serverového zařízení a zápis dat do něj.

Tato práce se zabývá samotným protokolem Modbus a jeho specifikací se zaměřením na variantu RTU implementace na sériové sběrnici, která je využita při řešení. Ostatní verze protokolu (ASCII a TCP/IP) jsou taktéž zmíněny, ne však v rozsahu v jakém je popsána RTU varianta. Jsou uvedeny algoritmy důležité pro vývoj samotné knihovny, jako je stavový diagram operace Modbus RTU serveru a algoritmus výpočtu CRC hodnoty používané k detekci chyb při přijímání zpráv.

Práce také popisuje programovací jazyk Forth. Jedná se o velmi netradiční jazyk a v práci je proto popsán podrobně včetně ukázky kódu a vnitřní struktury slovníku který Forth systémy používají pro ukládání funkcí (slov). Jsou také popsány vlastnosti jazyka důležité pro vývoj samotné knihovny: vektorové spuštění slov a modifikace kompilátoru za pomocí vytváření definujících slov. Práce obsahuje popis a ukázku zásobníkové aritmetiky a předávání parametrů které jsou integrální součástí operace Forth systémů. V práci jsou taktéž v krátkosti popsány kompilátory Forth jazyka pro mikrokontroléry: FlashForth, AmForth, 328eForth a ESP32forth. Z kompilátorů byl zvolen FlashForth pro svou schopnost definovat přerušovací rutiny čistě v jazyce Forth a pro podporu čipu Atmega328p přítomného na zařízeních Arduino UNO a NANO.

Značná část textu práce se zabývá samotnou výslednou knihovnou a jejím vývojem. Knihovna byla vyvinuta za pomoci desky Arduino UNO, na kterou byl nahrán modifikovaný systém FlashForth s uvolněným časovačem pro potřeby knihovny.

Tento modifikovaný systém byl zkompilován pomocí kompilátoru XC8 programu MPLAB ze zdrojových souborů obsažených v distribuci systému FlashForth. Knihovna byla otestována také na desce Arduino NANO, která je poté využita pro konstrukci demonstrační jednotky. V práci je popsána struktura zdrojových souborů a také styl zápisu kódu obsaženém v nich. Sekce práce jsou věnovány jednotlivým oblastem knihovny jako je zpracování zpráv, způsob volání funkcí (slov) na základě funkčních kódů obsažených ve zprávě či nastavování časovačů. Popsáno je také mapování paměťových oblastí protokolu Modbus.

Při vytváření knihovny byly použity materiály společnosti Modbus organization, popisující protokol Modbus stejně jako dokumentace k mikrokontroléru Atmega328p společnosti Atmel.

Během vývoje byl předefinován hlavní způsob řízení stavu server zařízení ze stavového automatu na řízení vnořenými cykly, jelikož stavový automat kladl vysoké nároky na obsluhu přerušením, vedl k časté modifikaci vektorů pro spouštění Forth slov a byl oběcně komplikovaný pro účely ladění. Implementace kontroly stavů pomocí vnořených cycklů je rychlejší a neklade takové nároky na přerušení generované časovači. Obě tato řešení jsou v textu práce popsána.

Při vývoji byla funkčnost ověřována za pomocí Modbus master emulátoru Modpoll. Byl také využit program Wireshark pro kontrolu USB packetů obsahujících zprávy cestujících do zařízení a z něj. Takto byly diagnostikovány chyby v obsluze jednotlivých požadavků master zařízení.

Byla zhotovena demonstrační jednotka skládající se ze tří Arduino NANO desek, popisu této jednotky je věnována poslední kapitola této práce. Je udeveno také elektrické schéma pro celou jednotku.

Cílů práce tedy bylo úspěšně dosaženo a výsledná knihovna je funkční. Výsledná knihovna se sestává ze 3 souborů které je možné interpretovat na zařízení se systémem FlashForth. Tyto soubory jsou strukturovány tak, že vyšší logika knihovny nezávislá na použitém mikrokontroléru je ponechána ve zvláštním souboru a při zavádění knihovny na čip jiný než Atmega328p je potřeba předefinovat pouze funkce (slova) obsažená v souboru specifickém pro mikrokontrolér Atmega328p a mapování Modbus paměťových oblastí. Samotná knihovna je po modifikaci použitelná s jakýmkoli zařízením se systémem FlashForth. Je možné ji použít i pro jiné Forth systémy, její portování pro jiné implementace jazyka Forth by však bylo značně náročnější. Součástí příloh je také soubor s ladicími funkcemi použitý při vývoji. Demonstrační jednotka prokazující funkčnost knihovny byla taktéž úspěšně zkonstruována.

Všechny zdrojové soubory pro knihovnu a demonstrační jednotku jsou obsaženy v elektronické příloze A.

KOUŘIL, Jakub. *THE LIBRARY FOR MODBUS RTU IN FORTH LANGUAGE*. Brno: Brno University of Technology, Faculty of Electrical Engineering and CommunicationF, Department of Control and Instrumentation, 2022, 47 p. Bachelor's Thesis. Advised by prof. Ing. Pavel Jura, CSc.

Author's Declaration

Author:	Jakub Kouřil
Author's ID:	220992
Paper type:	Bachelor's Thesis
Academic year:	2021/22
Topic:	THE LIBRARY FOR MODBUS RTU IN FORTH LANGUAGE

I declare that I have written this paper independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the paper and listed in the comprehensive bibliography at the end of the paper.

As the author, I furthermore declare that, with respect to the creation of this paper, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll. of the Czech Republic, Section 2, Head VI, Part 4.

Brno ....23.5.2022....   ....

author's signature*

*The author signs only in the printed version.

ACKNOWLEDGEMENT

I wish to show my appreciation to prof. Ing. Pavel Jura CSc. for taking the role of the supervisor of this thesis.
I would like to thank my consultant, Ing. Vilém Karský for his enthusiastic support and valuable input during the creation of this thesis and the development of the resulting library.
I would also like to extend my gratitude towards the bands Nightwish and Rammstein for making the countless hours spent working away at this project more enjoyable.

Brno 23.5.2022 ....

author's signature*

*The author signs only in the printed version.

Contents

Introduction **12**

1 MODBUS **13**

1.1 MODBUS server data model . . . 14
1.2 MODBUS over serial communication . . . 15
1.2.1 MODBUS RTU . . . 15
1.2.2 MODBUS ASCII . . . 18
1.3 MODBUS TCP/IP . . . 18

2 Forth **19**

2.1 Forth operation . . . 19
2.1.1 Interpreter . . . 20
2.2 The Dictionary . . . 21
2.2.1 Dictionary entry . . . 21
2.2.2 Defining Words . . . 22
2.3 The stacks . . . 23
2.4 Forth libraries . . . 25
Forth compilers . . . 26

3 Forth compilers **26**

3.1 FlashForth . . . 26
3.2 AmForth . . . 26
3.3 328eForth . . . 27
3.4 ESP32forth . . . 27

4 Forth MODBUS Slave Library **28**

4.0.1 Stylistic choices and appearance of the code . . . 28
4.1 Structure of the library . . . 30
4.2 Communication . . . 30
4.3 The workings of the library . . . 31
4.3.1 Message handling . . . 31
4.3.2 Slave device state control . . . 34
4.3.3 Timer management . . . 36
4.4 Supported function codes . . . 37
4.5 Modbus memory space mapping . . . 37
4.6 Verifying the functionality of the library . . . 38

5 Demonstration unit **40**
5.0.1 Code for the server units 40

Conclusion **43**

Bibliography **44**

List of appendices **46**

A Contents of the digital appendix **47**

List of Figures

1.1 MODBUS function codes 13
1.2 MODBUS RTU frame 16
1.3 MODBUS RTU message handling diagram 16
1.4 MODBUS RTU CRC algorithm 17
1.5 MODBUS ASCII frame 18
2.1 Hello world word execution 20
2.2 Forth dictionary entry example 21
2.3 FlashForth stack arithmetic example 24
4.1 modpoll successful write operation 39
5.1 Electrical schematic for the demonstration unit 42

Introduction

This thesis pertains to the usage of the Forth programming language in microcontroller software development with the goal to create a library of forth words implementing Modbus RTU slave/server unit behaviour. The devices utilising this library then could be used in applications primarily focused on small-scale automation and data collection projects.

The Forth language itself is a minimalist programming language used primarily in embedded systems. Its simplicity and other properties such as the ability to readily modify and extend the program on the go simply by connecting to the device via a terminal make it an excellent tool for many hobby applications as well as making it a valuable learning tool. Forth has however also been used in many commercial applications because of its small size and the execution speed of Forth code. NASA for example has used Forth systems on several of their projects [9] [10]. FlashForth, a Forth implementation for PIC and Atmega microcontrollers was used during the development of the library.

Modbus was chosen as a suitable communication protocol, owing to its simple nature suitable for hobby implementations and the fact it is an open protocol having documentation and implementation guides openly available. A host of Modbus master implementations also exist, providing easy access to a way to communicate with the device running the library.

This document contains the theoretical section spanning the first three chapters containing the descriptions of the Modbus protocol (with emphasis on the RTU variant of the serial line implementation), the Forth programming language and the basics of Forth system operation. The various Forth compilers are also described with FlashForth being described in further detail. A chapter is dedicated to the resulting library, describing its workings as well as the process of its development in detail. The last chapter deals with the construction of a small demonstration unit using Arduino NANO boards running the FlashForth system and making use of the library, presenting its funtionality.

1 MODBUS

This section is built upon the information contained within the protocol specification documents [1], [2].

MODBUS is an industry communication protocol created in 1979 by Modicon. Modicon was later acquired by a company known today as Schneider Electric. [7] In 2004 the Modbus organization was created to manage the protocol, congregating information about the protocol and providing implementation guides as well as links to useful resources pertaining to the protocol on their website.

MODBUS itself is an easy-to-implement, open communication protocol. Many manufacturers of industry automation devices use MODBUS as one of the available interfaces for their devices since the cost to do so is minimal with no licensing fees or royalties. With MODBUS not being a proprietary communication protocol it is also often used to interconnect devices from multiple manufacturers and it became a standard for Fieldbus communications. For the year 2020 roughly 10% of newly installed devices in factory automation were installed on a MODBUS network (with 5% for both MODBUS RTU and MODBUS TCP/IP) [8].

The MODBUS protocol encompasses two major variations, the MODBUS over serial communication and MODBUS TCP/IP. MODBUS over serial-line also provides two transmission modes, RTU and ASCII detailed in their sections below.

The protocol itself is a structure providing the connected devices with an ability to exchange data via a client-server communication. The protocol data unit (PDU) contains a function code spanning one byte and a data field of variable size, depending on the used function code. This PDU can then be expanded by adding more fields for server device address, error checking, etc. creating an application data unit (ADU), which is bus-specific.

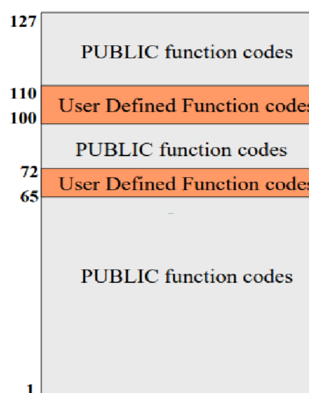

Fig. 1.1: Mapping of the MODBUS function codes [1]

The MODBUS function codes range from 0 - 255 as allowed by the allotted 8 bits. Code 0 is however not valid and the upper half of the range is reserved for exception responses, leaving the range of 1 - 127 for function codes, the available function codes can be seen in the figure 1.1 . Most of these codes are Public function codes with pre-defined behaviour described by the Protocol Specification. Function code ranges of 65 - 72 and 100 - 110 are reserved for User functions and are implementation-dependent or left unused. Some of the function codes are also reserved for legacy applications still in use by some companies, for detailed list see the Protocol Specification, Annex A. [1]

Not all of the public function codes have to be supported by any given MODBUS device, the selection of which function codes to support is application-specific. Some of the basic function codes to be supported by the device are the four data read codes displayed in the table 1.1.

function code	function
01	Read Coils
02	Read Discrete Inputs
03	Read Holding Registers
04	Read Input Registers

Tab. 1.1: Table of example function codes

The data field in a message from a MODBUS client contains additional information for the server, such as addresses to read from/write to, quantity of items to be read/written or subcodes specifying the client's request further. The field can also be empty in case the function code does not call for additional parameters.

A regular MODBUS server response message contains the original function code and the data requested by the client in the data field. If an error occurs during the handling of the request, the server sends a response containing an error function code (the original code with highest bit set to '1') and the data field represents the exception code. MODBUS exception codes are listed in a table in the Protocol Specification. [1], (pg. 48,49).

The size of a MODBUS PDU is limited to 253 bytes in order to ensure backwards-compatibility with the first MODBUS serial line implementations.

1.1 MODBUS server data model

MODBUS protocol differentiates between 4 data spaces. The mapping of said spaces can however overlap in the server device memory. The **Discrete Inputs** represent

read-only elements spanning a single bit, such as digital device inputs. The **Coils** are individual bits that can be both read from and written to, usually representing device digital outputs. **Input Registers** and **Holding Registers** and both the size of a WORD (2 bytes) with the Input Registers being read-only for storing of digitized analog values or for grouping of bit-sized elements. Each of these data spaces can contain up to 65536 elements addressed from 0 to 65535 as per the protocol specification. The exact size and layout of the data space is however implementation-specific and not all four data spaces have to be implemented.

1.2 MODBUS over serial communication

MODBUS networks are realized using a Master-Slave architecture on serial lines such as RS232 and RS485, with a sole master device representing the MODBUS client and multiple slave units working as servers. The slave devices do not communicate outside of responding the the master's requests and the master can thus control the traffic on the bus.

The serial line PDU contains an address and error checking fields on top of he MODBUS PDU. The master can send a broadcast message to all the slave devices by using the address of '0' in the request PDU, the slaves cannot respond to a broadcast message, doing so would result in congestion on the bus with all the slave devices trying to access it at once.

MODBUS over serial communication offers two transmission modes, RTU and ASCII, detailed below.

1.2.1 MODBUS RTU

The RTU transmission mode is the default mode for MOBUS devices communicating over serial buses and it has to be supported by each MODBUS device for use on a serial bus. In this transmission mode each byte represents two hexadecimal characters. The message frame can be seen in figure 1.2. The individual bytes of the message are transmitted over the serial bus using 11 bits. CRC (Cyclical Redundancy Check) is generated for each transmitted message and spans the last two bytes of the transmission. In case the CRC the recipient of the message calculates differs from the one contained within the transmission itself, an error specifying message corruption is generated.

As per the *Specification and Implementation Guide for MODBUS over serial line*, the delay between frames must be at least 3.5 character times with individual bytes within the frame being separated by no more than 1.5 character times. In case a byte within a single frame is delayed by more than 1.5 character times but

Slave Address	Function Code	Data	CRC
1 byte	1 byte	0 up to 252 byte(s)	2 bytes CRC Low CRC Hi

Fig. 1.2: MODBUS RTU message frame [2]

no more than 3.5 character times, the entire frame is declared as incomplete and ignored by the receiving device.

The document also contains a state diagram describing the process of handling MODBUS messages by the RTU devices, the diagram can be seen in figure 1.3

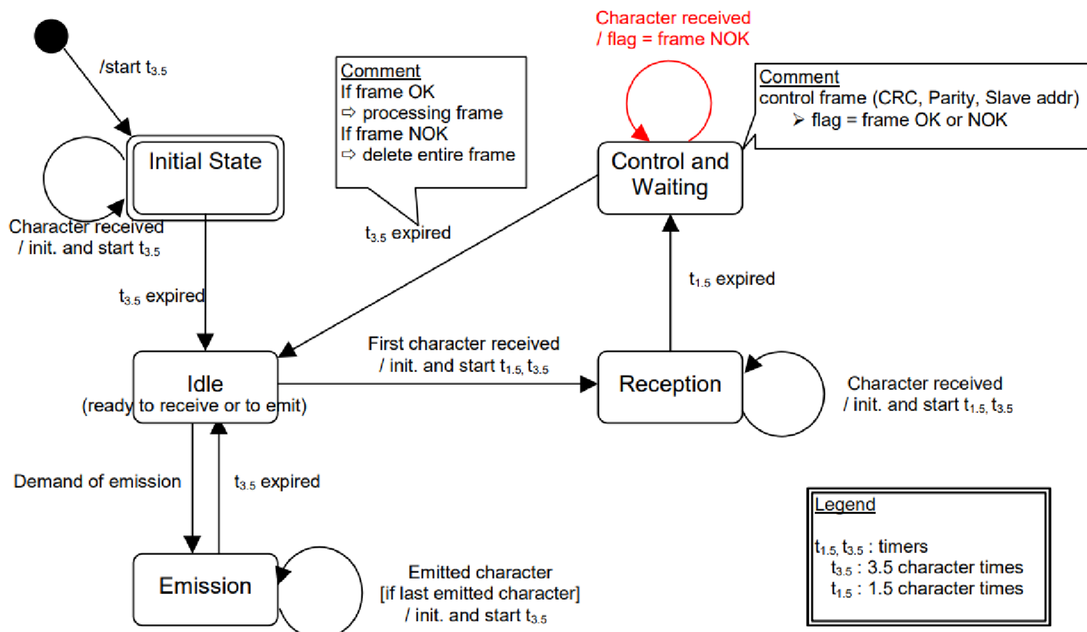

Fig. 1.3: MODBUS RTU message handling diagram [2]

CRC

MODBUS RTU uses a CRC (cyclical redundancy check) system in order to detect message corruption. The CRC field spans the last two bytes of any MODBUS RTU message.

The receiving device calculates the CRC value based on the bytes of the received message and compares it to the actual CRC value received as part of the trasmission. In case the two values are not equal, the message is declared corrupted and discarded.

The CRC itself is calculated using an algorithm described within the *Specification and Implementation Guide for MODBUS over serial line.* A calculation is executed for each of the transmitted bytes.

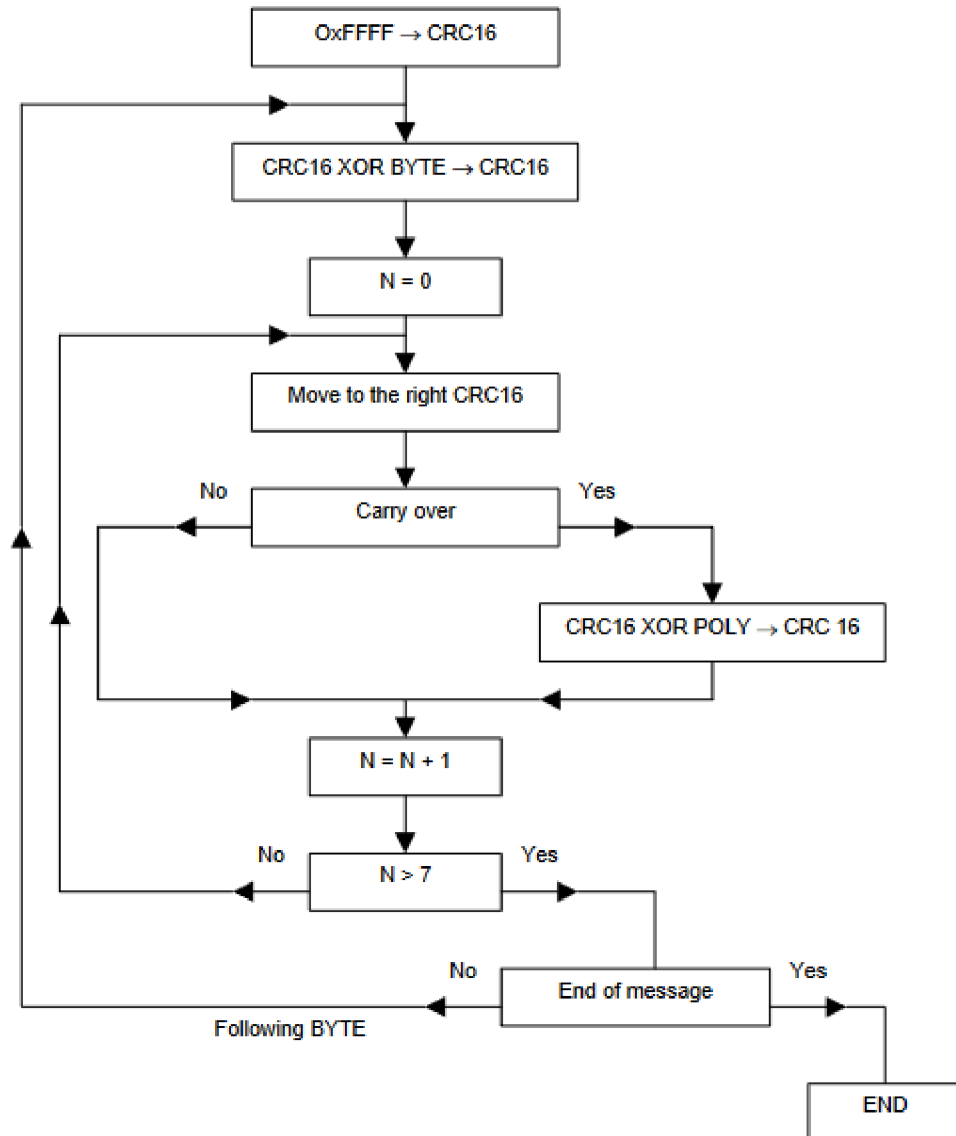

Fig. 1.4: MODBUS RTU CRC algorithm [2]

The algorithm is shown in figure 1.4 with **CRC 16** being a 16-bit register used to store the intermediate results between operations and **POLY** marking a constant CRC value of 0xA001. **BYTE** marks the individual bytes of the MODBUS message.

1.2.2 MODBUS ASCII

In the ASCII transmission mode, the messages are sent as a stream of ASCII characters delimited by colon as a character indicating the start of a transmission and CR/LF character marking the end of the message. This transmission mode is less efficient than RTU with each byte of the message being represented by two characters (thus making each byte twice as long to transmit) and the ASCII mode is thus used for devices unable to process faster RTU communication. The data frame for the ASCII transmission mode is displayed in the figure 1.5

Start	Address	Function	Data	LRC	End
1 char :	2 chars	2 chars	0 up to 2x252 char(s)	2 chars	2 chars CR,LF

Fig. 1.5: MODBUS ASCII message frame [2]

1.3 MODBUS TCP/IP

This variant of the MODBUS protocol encapsulates the MODBUS PDU within a TCP/IP packet to be sent over an Ethernet network. This allows MODBUS devices to be connected into and to communicate over an existing Ethernet network alongside other non-MODBUS devices, making the cost to implement such communication minimal when an existing Ethernet network is present. While useful, this version of the protocol is not the focus of this work and any further mentions of the MODBUS protocol pertain to the serial-line version of MODBUS (RTU variant specifically).

2 Forth

This chapter contains information sourced from the book Starting FORTH [3]

Forth (or FORTH) is a programming language developed in 1970 by Charles H. Moore. The driving principles behind Forth are simplicity and the ability to add/-modify code *'in situ'* without the need to recompile the entire program. Another advantage of Forth systems is their size, with simplicity of the system comes small space needed to implement it. The language itself consists of individual words, the definitions of which is kept in a dictionary. A Forth program is created by defining words in a hierarchical manner, with higher-level words combining previously defined words to achieve more complex tasks. The user program itself then becomes the highest-level word within a given project.

By building the application from smaller words it also becomes easier to individually test and debug them before combining their functionality in other, higher-level words. This can significantly speed up the development of embedded software. Some of the low-level words can also be defined directly using ASSEMBLY language further speeding up the execution, but tying the Forth system to the processors whose ASSEMBLY was used.

The simplicity of the programming language along with its readability also makes Forth a good choice for beginner programmers to learn on (though the Reverse Polish Notation takes time to adjust to). As Forth operates very closely with hardware it lends itself to people learning the basics of embedded systems as well.

Forth is primarily used in embedded applications, especially those closely interacting with hardware. It is suited for those applications due to the small size of the Forth system and the speed of execution it offers. NASA particularly has used Forth in multiple applications over its history. [9] [10]

2.1 Forth operation

A terminal is used to communicate with a Forth system. The user types commands for the Forth interpreter, which searches the dictionary for words with the typed identifier to execute them. If no word is found the input is assumed to be a number and pushed onto the Parameter Stack. A select number of commands switch to compilation time, e.g. commands beginning with a colon denote a new word definition and the new word is compiled and a new dictionary entry is created.

The dictionary is searched from the newest entry to the last, allowing for a word to be re-defined simply by creating a new entry (word) with an identical identifier as it will be found and executed first. The ability to add Forth words through a

terminal without the need to recompile the entire program makes it easy to modify existing code and to extend it.

To accomplish this a Forth system runs a cycle handling the input stream from a terminal. The word **QUIT** accepts the data passed though the terminal and calls the word **INTERPRET** to handle the input string. **QUIT** can thus be used to terminate the execution of an application and return to processing the input from the terminal.

Code example: a 'hello world' program written in Forth can be written as follows:

Listing 2.1: Forth hello world word

```
: HELLO_WORLD ." Hello world. " ;
```

The interpreter will create a new dictionary entry for a word called **HELLO_WORLD** which will use the word **."** to print a string of characters to the terminal, the **;** word finalises the creation of a new definition and exits the compilation mode. (Both the interpreter and the dictionary are described in further detail in the following subsections)

The defined word can then be executed by calling its name from the terminal, doing so on a FlashForth system produces a result visible in figure 2.1. (FlashForth also appends the information that the command was executed successfully, the current contents of the parameter stack and the selected memory type)

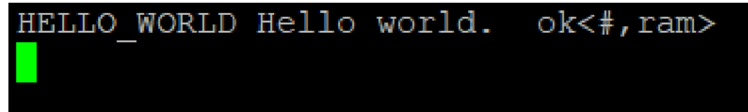

Fig. 2.1: Hello world word execution

2.1.1 Interpreter

Forth systems usually contain two interpreters, the text interpreter and address interpreter.

Forth text interpreter takes a sequence of words the user entered through the terminal using the word **INTERPRET** which executes the word in case a definition is found in the dictionary using the word **EXECUTE**. If no such dictionary entry is found, the input is parsed as a number according to the current set number base and pushed onto the Parameter Stack.

The address interpreter is used when a word in the dictionary is called, using **EXECUTE**. It sequentially executes the words/code (once again using the address interpreter) specified by the executed word in the code field.

Vectored Execution

By storing an address of a word inside a variable we can use the word **EXECUTE** to execute the code of the addressed word. The stored word address can then be changed later on, altering the behaviour triggered by executing the original variable. This mechanism can be compared to a pointer to a function. Variables containing word addresses for vector execution are conventionally named with first character being ' as the **'** word [*tick*] is used to retrieve a word address ([**'**] has to be used inside of a definition instead.

2.2 The Dictionary

The Forth dictionary contains the definition of every word in any given Forth system. A new entry is created using a defining word. Data structures like variables and arrays are also stored as dictionary entries. Dictionary entries can be organized into vocabularies and the order in which the individual vocabularies are searched can be changed.

The dictionary takes up most of the memory in the Forth system.

2.2.1 Dictionary entry

The dictionary entry has the following structure:

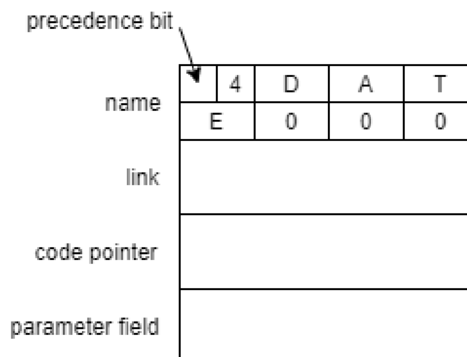

Fig. 2.2: Forth dictionary entry example [3], redrawn to achieve satisfactory image quality

Forth systems can differ in the number of dictionary entry fields and their order, the four basic field are shown in the example entry 2.2 .

The name field contains the ASCII character identifier of the word, though a number of characters and only the first X characters of the word name can be used instead (this may lead to unintentional re-definition of words and should be looked out for). A precedence bit is also stored as the first bit of the name field, specifying whether the word inside the dictionary entry should be executed during compilation or not, the significance of this is detailed in the Defining Words section further.

The example shown in figure 2.2 uses the first 7 characters and a the total number of characters in the name field (as well as a precedence bit), as the words used in the example is **DATE**, only four of the 7 stored characters are used and the rest is padded with zeros.

The dictionary is a singly linked list and the link field is used to store the address of the previous word in the dictionary.

The code pointer/code field contains either the address of the run-time code to be executed when the word is executed or the code itself. This can be the code for handling variable entries, constants, the code for user-defined words set to execute the words listed in the data field, etc.

The data field is of varying size depending on the dictionary entry. For variables it contains the actual value of a variable, as it does for arrays. For user-defined words it contains the list of addresses of words comprising the definition with the last address being of the word **EXIT** which stops the execution of the word and returns the execution to the word, the address of which is stored on the return stack.

2.2.2 Defining Words

Defining words in Forth are used to create new dictionary entries. Defining words are marked by a precedence bit in their dictionary entry. A number of different defining words are implemented, such as **:** for creating 'colon definitions' for defining a new word, or **VARIABLE** for creating a new variable. Defining words can also have different behaviour depending on whether the system is in compile-time or interpret-time, e.g. **VARIABLE** will create new dictionary entry for a variable during compile time, but it will instead push the address of a specified variable onto the Parameter Stack during interpret-time.

The definition of the defining word is split into two parts, one defining the compile-time behaviour and the other specifying the run-time behaviour of the word. These two sections are delimited using the word **DOES>** in the manner that the compile-time code precedes the word **DOES>** and the run-time code follows it.

New defining words can be implemented by the user to for example create more complex data structures needed for specific application. The Forth compiler can be extended in this way.

Compiling words also exist, while not making new dictionary entries, they still possess different code to execute depending on whether the system is in compile-time or run-time. These words are usually used inside a colon definition while defining a new word, for example for filling out the data fields of the newly created word definition in the dictionary.

2.3 The stacks

Forth uses several stacks. The most important one is the Parameter/Data Stack (or simply the Stack as it is the most used one) used to pass parameters to words and to execute arithmetic operations. Forth uses Reverse Polish Notation (RPN) so all arguments (and operands) have to be pushed onto the stack prior to executing a word requiring said parameters. The words will use the parameters and can also push a return value onto the stack.

It is this principle that makes the Forth code fast to execute, the stack-oriented operation and RPN are efficient and close to how the processor handles operations. It also makes each words naturally re-entrant, pushing the parameters onto the stack and executing the word will not lead to problems even when the same word is mid-execution (unless the stack should run out of space).

Words should not leave their parameters on the stack after they are executed to prevent accidental leaks leading to the stack growing over time and eventually overflowing. It has become a standard to define the expected parameters and return values for any given word using the following notation:

$$(P—R)$$

where P is an ordered array of parameters to be passed into the word and R is an ordered array of its return values. This notation essentially describes the state of the stack before executing the word and the state after the execution concludes. A word described by $(n1\, n2—n3)$ would expect two numbers to be passed as parameters through the stack and would leave a number on the stack as a return value (it is also customary to define the type of said parameters and return values, n = integer, d = double, ...). Forth provides a set of words manipulating with the values on the stack. Most of these words are however limited to affecting only to the top three values at most so there should not be too great a number of values being handled on the stack at any given moment.

The items on the stack occupy either one or two stack positions, with the latter being double-cell numbers. In case of double-cell numbers, the most-significant cell is stored on top of the least-significant on the stack. The words in Forth that work with double-cell numbers are conventionally prefixed with '2', such as **2OVER** to distinguish them from their counterparts operating on single-cell numbers.

The Parameter stack can be used to carry results from one words as parameters for another, eliminating the need to define variables to temporarily store data between function calls in conventional programming languages.

The Return Stack is used to store return addresses when entering a lower-level word, much like call stacks in other programming languages. It can also be used to temporarily store values within a single defined word. Loop control stack also exists in order to store the start and stop indexes of Forth loops, it can however be stored on the Return stack.

Forth stack arithmetic example:

Listing 2.2: Example of Forth stack arithmetic

```
12 5 + 14  -
```

Forth will interpret the numbers being passed through the terminal as numeric values and push them onto the parameter stack, the + and - words will consume the topmost parameters on the stack and push the result of the operation back onto the stack.

An example of this simple stack arithmetic operation sequence in listing 2.2 performed on a system running FlashForth can be seen in figure 2.3. Since FlashForth lists the parameter stack contents after each operation is performed, each number and operation is performed individually to improve clarity.

```
  ok<#,ram>
12  ok<#,ram> 12
5  ok<#,ram> 12 5
+  ok<#,ram> 17
14  ok<#,ram> 17 14
-  ok<#,ram> 3
```

Fig. 2.3: FlashForth stack arithmetic example

2.4 Forth libraries

Forth compilers can support a single microcontroller only up to a range of microcontroller families, depending on the compiler. Given how close to hardware Forth operates (for the purpose of speed of execution of Forth programs) it can be difficult if not outright impossible to develop a universal library for each of the devices supported by the given Forth compiler (if the compiler does in fact support multiple microcontroller at all). To combat this issue, Forth libraries can be hierarchically split into several libraries, grouping up device-specific code into standalone 'low-level' library and the high-level logic into another library.

In this way the user only has to re-implement the lowest level library himself in order to make use the original higher-level library. The device-specific code may then not be included with the library at all, leaving only a description of functions with names expected by the existing library for the user to implement on his own device.

Forth systems themselves can work similarly, eForth for example requires around 30 words to be implemented using code specific to the target device, with the rest of the system being device-agnostic and built upon these 30 basic words.

3 Forth compilers

A number of Forth implementations have been created over the course of Forth's history, Forth Interest Group (FIG) keeps a list of notable non-commercial Forth compilers on their webpage (as well as a list of commerical Forth system vendors). [11]

Following are the sections dedicated to the individual Forth compilers:

3.1 FlashForth

This section is based on the information contained within the FlasForth Guide. [5]

Flashforth is a Forth compiler for the PIC18F, PIC24, PIC30 and PIC33 as well as the Atmel Atmega microcontrollers. It allows the user access to all types of memory: RAM, EEPROM and FLASH. FlashForth provides words for switching between memory types and it also states current memory type after the execution execution of every command.

FlashForth is a robust system striving for stability. In order to preserve a working core of the system that the user can revert to in a case of need, it prevents the user from executing potentially harmful code, such as redefining a word not located in the user dictionary or overwriting the FlashForth Kernel.While this makes the system more stable it also denies the user access to some of the FORTH programming tools, such as redefining existing words by creating a new definition further down in the dictionary. FlashForth also provides the words FL+ and FL- to allow/prevent attempts to write to FLASH or EEPROM memories, these words can be used in an application to 'lock' the FLASH and EEPROM outside of words dedicated to storing data in these memory types.

A turnkey word is provided, allowing for a word to be executed upon startup of the device.

FlashForth also provides the ability to write interrupt routines in FORTH, abstracting the user from the need to write in ASSEMBLY entirely.

3.2 AmForth

The following section contains information from the AmForth Technical Guide. [12]

AmForth is a Forth compiler for the AVR ATmega microcontrollers adhering to the Forth ANS94 standard. AmForth, similarly to FlashForth allows for the development of interrupt routines directly in Forth language. Most of Amforth application resides within the flash memory of the device, with variables and stacks

being stored in the RAM. AmForth also provides the ability to generate and handle exceptions.

Given that the AmForth system is written using Assembly language, some AmForth words in user applications may need to be defined in Assembly by the user. An example of this can be found in the documented Amforth project. [4] This may potentially make the development of a Forth library using Amforth troublesome, as the system itself might need to be modified in order to make use of the final library.¨

3.3 328eForth

This section is built upon the defining document of 328eForth. [6]

328eForth is based on the eForth standard, a minimalistic Forth implementation created by Dr. C.H. Ting, a promoter of Forth language. As such it is highly optimised and tied to a single microcontroller.

This Forth compiler for the Atmega328P microcontroller (present on the Arduino UNO and NANO boards) was created as a way to get closer to the bare metal than one would with regular C language. It was also created in order to interactively teach people the basics of the Forth language and the Atmega328P workings, so a host of examples and learning lessons are available for the system. The documentation for this compiler is also extensive.

328eForth does not allow for the creation of interrupt routines using Forth and does not provide the ability to work with the EEPROM memory.

3.4 ESP32forth

This section is based on the information contained on the ESP32forth website. [13] ESP32forth is a Forth compiler for a variety of ESP32 boards. It allows the user to modify and work with the Forth system by using Arduino IDE. Additional words can be defined using C functions.

ESP32forth also provides a degree of support for floating point numbers, which may be relevant to some applications. This feature is currently work in progress though.

This compiler also provides the ability to handle interrupts using wholly Forth words similarly to FlashForth.

4 Forth MODBUS Slave Library

The entire library was developed on an Arduino board with Atmega328p microcontroller running the FlashForth system. A build of FlashForth with CPU load measurement disabled must be used in order for the library to function properly for reasons further explained in section 4.3.2.

The source files as well as all associated files are included in the digital appendix A.

Code modifications within this chapter

The original source files were written with only the limitations of the TCL terminal emulator provided with FlashForth distribution in mind. This was the primary way of interfacing with the FlashForth system during development, providing a way to automatically interpret entire files instead of manually interfacing with FlashForth each time the library words need to be refactored or fixed. Since FlashForth does not allow for duplicate word definitions, the original word needs to be removed using the word **marker** before interpreting the definition again. This would be quite laborious to do by hand and the emulator played a big part in the process of the development.

Said emulator allows for up to roughly 80 characters per line, so any comments or lines approaching this length were broken up over several lines. This still leads to some issues while displaying the code within this document and thus any Forth code contained in listings inside of this chapter may have been modified in order to better fit the page, such as wrapping comments that are too long over to the next line.

In the source files, I have chosen to write logical expressions on standalone line before a branching word, such as **if**, which consumes the produced flag. In some instances, the code examples have been modified so that the branching word is included on the line with the logical expression so save space taken up by the listing, such as with listing 4.5.

Certain comments or their parts are also entirely omitted in this document as opposed to the actual source files. This is done both to cut back on some of the more detailed explanations and mainly to remove comments that are in length discussing issues covered by the scope of this document itself to prevent needless repetition of information already discussed.

4.0.1 Stylistic choices and appearance of the code

Although Forth systems are usually case-insensitive and Forth code is commonly written using only capital letters, FlashForth is an exception with it being case-

sensitive. Since essentially all predefined words are in lowercase, I have kept most of the word names lowercase as well, with a few exceptions to help readability.

The source files are thoroughly commented and described. The notation for stack operations described within section 2.3 was used with words that consume or produce values, but it is omitted with most of the words that do not modify the values on the stack in order to avoid unnecessary clutter. Comments at the beginning of a word definition that are indented twice pertain to the word as a whole while comments that are either indented on the same level as code or are written at the end of the line (in case the space left after the words on said line allows to do so) describe individual operation within the word.

As for the code itself, words that together constitute a logical operation are written on a single line to preserve the readability of the code. This aids to separate the words into clusters that are easier to comment as well and seems to be the default way to structure Forth source files used in FlashForth source files and many projects. In case of more complex words, they were often defined by first defining words handling sub-tasks of the final word, as is common during development using the Forth language. A clear example of this can be viewed in the section 4.3.1 describing CRC calculation.

The principles described above can be observed in code example listing 4.1.

Listing 4.1: Code and comment structure

```
: mbs_getDiscrete ( n --- f )
        \ reads specified discrete input and leaves flag
        \ on the stack equal to the value read
    mbs_diMap c@        \ read byte containing input bit
    and                 \ read input using mask
    0 <> ;              \ collapse masked value into flag
```

As for the naming of the words themselves, FlashForth uses the first several characters of a word's name as well as the name's total length as a unique identifier for the word (described in section 2.2.1 including an example). This led to some words having to be renamed over the course of the development. It is also the reason why most of the words begin with mbs_ or similar in order to try and prevent any naming conflicts with end user application.

Being fairly new to Forth programming, my own skills grew considerably over my time spent coding with it, this has lead to a certain degree of discrepancy of code quality between portions of the library written at different times. Though words were refactored during the development of the library, some still are containing sub-optimal code that could be further optimised.

4.1 Structure of the library

The FlashForth words the library consists of are split among three files:

- **library.fs**

  This file holds the majority of the words defined during the development. An effort has been made to make words contained within this file be system-agnostic and they should work on microcontrollers other than Atmega328p which was used during the development and testing.
- **Atmega328p.fs**

  Words in this file are specific to the Atmega328p and they would need to be re-implemented in their entirety should the library be used with another microcontroller as they pertain to the setup and handling of timers and timer registers.
- **config.fs**

  The Modbus server configuration words are contained within this file. The device ID, the communication speed which affects the timer intervals allowed between characters during transmission and memory mapping are handled by words from this file. The word defining device ID will need to be modified for each device connected to a single serial line.

In order to preserve dependencies of words, the files must be interpreted in a specific order when using a terminal to upload them to the device. The order should be the following:

1. config.fs
2. Atmega328p.fs
3. library.fs

A fourth file: **debug.fs** is provided alongside the library source files. This file contains words assisting with verifying the functionality of the library, such as pre-defined test Modbus messages. The words within proved essential during the development and may be useful when/if the library is tested on another microcontroller.

4.2 Communication

The MODBUS communication is making use of a serial port on the board running a Forth system, as such a way to control the traffic on the port may be needed on devices with only a single serial port used by the Forth terminal by default.

To this end, the words **emit+** and **emit-** are provided, with the former enabling the functionality of the word **emit** to transmit data on the serial port and the latter

disabling it. This is done using vector execution with the help of the predefined word **’emit**

Listing 4.2: Words emit+ and emit-

```
: emit- ( ---)
    \ disables the ability of the device
    \ to emit data onto the serial port
    ['] drop 'emit ! ;

: emit+ ( ---)
    \ enables the ability of the device
    \ to emit data onto the serial port
    ['] tx0 'emit ! ;
```

The definitions of words **emit+** and **emit-** can be seen in listing 4.2. Modifying the execution token stored in word **’emit** alters the behaviour of vectored word **emit**, with **emit-** using the execution token of the word **drop**. This makes it so that **emit** discards a value rather than actually transmitting it over the serial port. The word **emit+** then restores the original functionality of **emit** by having it transmit a character over serial port tx0.

By using the serial port, other system words could also be utilised, such as **type** which transmits contents of a specified buffer over the serial port or **key** and **key?** which facilitate reading characters from serial port buffer. It also comes with the limitation of using the port as-is with settings FlashForth uses, though these can be changed by recompiling FlashForth with modified values if necessary. By default the port is set to 38400 baudrate, 8 data bits and no parity.

4.3 The workings of the library

The following subsections pertain to the individual facets of the library and its development.

4.3.1 Message handling

An In-Out message buffer is used to store the received Modbus messages and to then modify them before transmitting the modified message as a response since Modbus responses echo the device address and function code (in case of a correct request).

A defining word was set up for the buffer and when it is interpreted later on it leaves the starting address of the buffer as well as the length of the currently stored

message (which is kept in the byte preceding the beginning address of the message itself, a technique of constructing data structures common to Forth) as words such as **type** use these parameters in said order. Deleting bytes also becomes as easy as decreasing the stored message length. The defining word as well as the creation of the buffer can be observed in listing 4.3.

Listing 4.3: Buffer defining word

```
: mbs_buffer ( --- addr n)
        \ sets up a buffer of a specified size, which
        \ will leave its address and current
        \ number of occupied bytes on the stack
    create
        0 c, allot
    does>
        1 + dup 1 - c@ ;

ram #256  mbs_buffer mbs_msgBuffer
```

A suite of buffer operation words is included within the library in order to help with common operations from the simple ones such as writing to individual bytes or bits or resetting the buffer to zero size to more complex operations like loading computed CRC values into the buffer or modifying the buffer contents to assemble an error response with given error code.

When a byte of a Modbus message is received, it is pushed into the buffer. The buffer is reset between each message.

CRC calculation

A word calculating the CRC value was defined using the algorithm from figure 1.4. First a **crc_cycle** word was set up, this word handles a single byte of the Modbus message, with all 8 associated shifts and XOR operations with a preset value. This way, the debugging of the words was significantly easier. The Modbus documentation [2] includes an example of the CRC value calculation algorithm with intermediate values in the CRC register listed, which came in useful during the debugging of the words. A wrapper word initializing the CRC value and iterating over each message byte was then defined. Both words are contained within listing 4.4.

FlashForth does not by default contain the definition for the words **DO** and **LOOP** which set up a cycle with loop index counting upwards from one specified value to another, implementing a **FOR ... NEXT** loop instead. This construction

has a loop index counting downwards to 0 from a passed value which unfortunately proves to be an obstacle when an index counting upwards is required. The **DO ... LOOP** cycle for FlashForth can be loaded from a separate file provided with the distribution, but I wanted the library to have no external dependencies and thus opted to calculate the index instead.

FlashForth uses the same little-endian way of storing data as the CRC value in Modbus messages, eliminating the need to swap the bytes and write them into the message individually and the final 16-bit CRC value can be directly written to the address of the first byte using the word **!**.

Listing 4.4: Words handling CRC calculation

```
: crc_cycle ( CRC16 --- CRC16' )
        \ an inner cycle handling the shifting and xor with
        \ preset value for a single byte of a MODBUS message
    $8 for
        dup %1 and
        \ examines the shifted out bit and proceeds
        \ to either xor with a preset value or carry on
        \ with the next shift
        swap 2/ %0111111111111111 and swap
        if
            $a001 xor
        then
    next ;

: mbs_crc ( msg_addr msg_length --- msg_addr msg_length crc )
        \ computes the crc for a specified MODBUS message
    $ffff \ crc value initialization
    over
    for
        \ loop index calculation
        over r@ 1 + - >r >r over r> swap r> + c@
        xor
        crc_cycle
    next
    r> drop ;
```

Function code execution token table

An array of execution tokens is created as a part of the library. Since it will only be modified during the programming of the device, it was placed into the flash memory in order to save 512 bytes of ram memory. Execution token for a word handling Modbus message with a given function code lies at the index of the array equal to the function code. At the point of creation of the array, it is filled with execution tokens of the default response sending error message stating that an unsupported function code was requested. The default response is vectored in case it needs to be modified in an application without the need to rewrite the values in the table itself.

Support for additional function codes can be implemented simply by defining a word to handle the messages with the new function code and then inserting its execution token at the appropriate index within the function table. Given the nature of Forth. This can also be done after the initial act of programming and deploying a device simply when support of new function codes becomes necessary.

4.3.2 Slave device state control

The original design of the library operation included a state machine. This idea was later abandoned in favor of a far simpler approach of using nested cycles, which proved easier to develop and debug as well as being quicker to execute.

The state diagram calls for the use of two timers. Atmega328p provides three timers: 8-bit timers **Timer0** and **Timer2** and then a 16-bit timer **Timer1**. [14] By default, **Timer0** is used by the FlashForth word **ms** to pause execution for a defined time period. **Timer1** is then used for CPU load measurement and **Timer2** is left unused. In order to make use of two timers for the library, the CPU load measurement must be disabled. FlashForth provides the ability to do so by altering the config-xc8.inc file. A new FlashForth build will then have the CPU load measurement disabled. It is also possible to have the **ms** word use a different timer, though this was not necessary for the development of this library.

A FlashForth image with the CPU load measurement disabled was compiled using the MPLAB program with the XC8 compiler from the source files from the FlashForth distribution. This file: **ff.hex** is included in the digital appendix A

Failed state machine

Originally, the reception and handling of the Modbus transmissions was supposed to be governed by a state machine implemented according to the state diagram described by the figure 1.3. This state machine was programmed as a word **mbs_main** which would cyclically call vectored word **mbs_state**. The word which is called by

mbs_state was swapped during timer interrupts as described by the state transition conditions in 1.3. The transition from Idle state was an exception in this regard as the transition happens after reception of the first transmission byte.

One of the transition handling words was set up as vectored word as well in order to break the cyclical dependency of the state machine words. This works akin to declaring a function prior to defining it in traditional programming languages like C.

This solution however proved to be impractical, requiring frequent modifications of the interrupt vectors and vectored words, costing processing time. It also proved challenging to properly debug and further develop and it was entirely scrapped later on. The source code can be seen as part of the digital appendix A.

States governed by cycles

A new, far simpler approach to controlling the slave device state was used, utilising three nested cycles. The outermost cycle represents the program loop itself. The first nested cycle is the *de facto* idle state of the server, awaiting the first message byte. The innermost nested cycle takes care of receiving the remainder of the transmission. Timers are used to set flags indicating the end of the message transmission and if the received message is valid, as described by the original state diagram in figure 1.3. The inner cycles are exited once the flag indicating the end of the message is set and the message is then processed before re-entering the outermost cycle and waiting for the following transmission. The cycles handling a single message can be viewed in listing 4.5

Listing 4.5: cycle handling messages

```
begin
    mbs_msgBuffer buffer_reset
    key? if
        key
        mbs_msgBuffer buffer_push
        1 timer15en     \ setup timers and prepare to receive
        1 timer35en     \ the rest of the message
        begin
            key? if
                key
                mbs_msgBuffer buffer_push
                timer15reset
                timer35reset
            then
```

```
        mbs_endedMsg c@ until
        mbs_msgBuffer buffer_addr_get \ check the device address
        mbs_device_ID c@ = if
            mbs_check_crc if
                mbs_process     \ process the received message
                map_register
            then
        then
    then
mbs_endedMsg c@ until
```

This approach also places little emphasis on the role of Timer1,5, using it only to mark the message as not valid as opposed to having it govern the state transitions like it was supposed to do with the previous design. It would thus be possible to remove the timer from the library entirely, should one desire to do so. This would allow the user to sacrifice the ability to detect message defects but would free up one of the timers for use within the final application. The CRC calculation would then be the only factor detecting transmission errors, but having a timer available might be worth doing so for certain applications. It may also be possible to use both the A and B compare registers of a single timer in place of using the A register of each of the two. This is something worth looking into in the future.

4.3.3 Timer management

The library currently makes use of two timers with **Timer2** representing **timer1,5** from Modbus documentation [2] and **Timer1** serving as **timer3,5**. Both timers run in Output Compare mode, with the value to compare to (written to registers OCRxA) being calculated with the help of a formula sourced from the Atmega datasheet [14] (page 101, CTC timer operation) and the period to time being given by Modbus documentation [2]. A word setting up each of the timers with regard to the serial port baudrate is provided. The value to store in the register is calculated using the following formula:

$$OCR2A = \frac{11000000}{baudrate} \cdot \frac{16}{2 \cdot prescaler} \cdot 1.5$$

$$OCR1A = \frac{11000000}{baudrate} \cdot \frac{16}{2 \cdot prescaler} \cdot 3.5$$

The first fraction represents the interval of 1 character transmission on a serial port in microseconds. The second fraction is based on the formula of timer interrupt frequency sourced as described above. The 1.5 and 3.5 constants are used as the

timers are supposed to time 1.5 and 3.5 character transmission times. Both timers run at prescaler of 256 in CTC mode.

These expressions were then simplified as to not waste time evaluating them each time the word is called. The final implementation used in the library then contains only a single division:

$$OCR2A = \frac{343750}{baudrate}$$

$$OCR1A = \frac{1203125}{baudrate}$$

FlashForth, while operating with 16-bit values normally, provides math words for certain 32-bit math operations, making this an easy process.

As per the Modbus serial line documentation [2], fixed values for the Output Compare registers are used if the communication speed exceeds 19200 baud in order to not generate excessive load on the CPU by creating interrupts in quick succession. For **Timer2**, the value 23 is used, while the value 55 is utilised for **Timer1**.

An interrupt word has been defined for each of the timers, with **Timer2** interrupt marking the currently received message as invalid and **Timer1** interrupt signaling the end of a message. Each of the timer interrupt words also disables the timer associated with the interrupt in order to prevent the needless generation of multiple interrupts while handling a single message.

4.4 Supported function codes

The library currently contains the support for 6 function codes:

- **Read Coils**
- **Read Discrete Inputs**
- **Read Holding Registers**
- **Read Input Register**
- **Write Single Coil**
- **Write Single Register**

These words operate using memory mapping defined in **config.fs**.

4.5 Modbus memory space mapping

Modbus memory spaces are mapped to physical memory with the words in **config.fs**. The memory space mapping can be redefined to fit any particular needs of the user application. The fact that Modbus registers are 16-bit and most of Atmega328p registers, such as port registers, are 8-bit introduces slight issues with mapping of the registers to physical memory.

In order to circumvent this issue, a separate 16-bit register is defined within ram memory, the contents of which can then be mapped to any address. As for the default configuration of the library, a word mapping the LSB of the register to port D data register and the MSB of the register to port B data register is provided. The word also avoids modifying the bits 0 and 1 of the D port as these are mapped to the RX0 and TX1 pins of the Arduino board and are utilised in communication with it.

All of the memory spaces span this register only by default, leaving the server with one holding/input register and 16 coils/discrete inputs. An example of mapping words can be observed in listing 4.6. For the words mapping single-bit values such as coils, the mapping words provide both a memory address as well as a bit mask, where the register mapping words only provide the address of the lower-end byte of the register, which FlashForth can use to write/read a 16-bit value value using the predefined words `!`/`@`.

The mapping words are also defined with modification in mind, the word register can be swapped to a starting address of any memory block allocated for Modbus memory space providing an easy way to modify the library to the needs of any particular application.

Listing 4.6: Memory mapping words

```
: mbs_diMap ( diIndex --- mask Addr )
        \ word mapping Modbus digital input address to a
        \ real address and bit access mask is provided
    #8 u/mod swap 1 swap lshift register rot + ;

: mbs_hrMap ( hrIndex --- Addr )
        \ word mapping Modbus holding register address
        \ to a real cell address
    cells register + ;
```

4.6 Verifying the functionality of the library

A Modbus master emulator sourced from www.modbusdriver.com/modpoll.html was used in order to communicate with the device running the library code and to verify its correct behaviour. This emulator supports all of the function codes the library recognizes, though it does not provide support for broadcast mode. The master also provides support for function codes 15 and 16 which write multiple coils and registers but are not supported by the library. By sending messages with these

function codes, the correct generation of error responses was confirmed. An example of the modpoll emulator use can be seen in figure 4.1.

```
jakub@jakub-Aspire-A715-72G:~$ modpoll -a 1 -m rtu -t 4 -r 1 -c 1 -o 0.5 -1 -b 38400 -p none
modpoll 3.10 - FieldTalk(tm) Modbus(R) Master Simulator
Copyright (c) 2002-2021 proconX Pty Ltd
Visit https://www.modbusdriver.com for Modbus libraries and tools.

Protocol configuration: Modbus RTU, FC6
Slave configuration...: address = 1, start reference = 1, count = 1
Communication.........: /dev/ttyUSB1, 38400, 8, 1, none, t/o 0.50 s, poll rate 1000 ms
Data type.............: 16-bit register, output (holding) register table

Written 1 reference.
```

Fig. 4.1: modpoll successful write operation, parameters *path to device* and *write value* cropped out due to size constraints

Wireshark was also used over the course of the development, especially in the later stages. It provided the ability to monitor and display the individual messages with the help of **usbmon** module. This proved exceptionally useful as any defects in messages could be quickly diagnosed and patched out by modifying the words responsible for the incorrect behaviour.

5 Demonstration unit

The demonstration unit consists of three Arduino NANO boards powered by a 5V power supply module (Which can be supplied by 12V from a transformer or via USB). Each of the Arduino units' serial port is connected to a RS485 bus via a transceiver module. The entire unit has been constructed on a breadboard, so that it can be later dismantled and the individual components used in other projects.

The RS485 bus is represented by a simple two-wire configuration with two of the breadboard channels being used as the bus itself. Since the project is contained on a small breadboard, this has proven to be a sufficient solution. Though for other projects and actual deployments of the devices with the bus spanning significantly larger distances a more robust implementation of the RS485 physical layer, such as CAT5 cable variant described by the protocol specification, will likely be needed.

A USB cable is connected to an additional transceiver and onto the RS485 bus. the emulated Modbus master requests are sent over this USB port using the Modpoll Modbus master emulator from a computer. Wireshark can also be utilized in case the messages themselves are to be observed.

Port B and D pins that are available for each of the units are connected to an LED with an appropriate resistor in place to limit the current. This serves as a simple way to indicating the Modbus register state. The LED used is a generic LED (D1) with forward voltage of 1.9 - 2.1 V and forward current of 20 mA. A 150 Ω resistor (R1) was utilised to limit the current through the LED.

The wiring diagram for the demonstration unit can be seen in figure 5.1.

5.0.1 Code for the server units

The units are running code using the library, with a word mapping the defined register in ram to ports B and D injected into the main cycle of Modbus operation in order to avoid the need to define an entire new task within FlashForth solely for this operation.

using the port C instead of the D port was originally considered, but ultimately decided against in order to make the demonstration unit more compact as port C pins are on the opposite side of the board than the port B pins, requiring a larger breadboard to properly connect to the resistors and LEDs.

A unique device ID is written to each of the device by storing the unique ID within the appropriate variable. The IDs used are 1, 2, and 3 for simplicity.

A word is defined, which sets up both ports B and D as output upon startup, with the exception of the bits of port D used by the serial port as mentioned in the previous chapter. Main word is then defined, calling mentioned setup word before

entering the loop of Modbus server operation. Both of these words are contained within the file **application.fs** which can be viewed in listing 5.1. The modified library file **library.fs** is included in the digital appendix A. The files **config.fs** and **Atmega328p.fs** were left unchanged for the purposes of the demonstration unit.

Listing 5.1: Application words

```
: app_setup
        \ set both ports as output
        \ and initialize register
   $ff DDRB c!
   %11111100 DDRD mset
   $0000 register ! ;

: main
        \ sets up the application and runs the
        \ Modbus server
    app_setup
    mbs_main ;
```

The main word is then declared the turnkey word by interpreting a command shown in listing 5.2. With doing this, the device programming is finished. The device will execute the main word after each startup. Each of the Arduino units thus behaves like a simple remote output module.

Listing 5.2: defining turnkey word

```
' main is turnkey
```

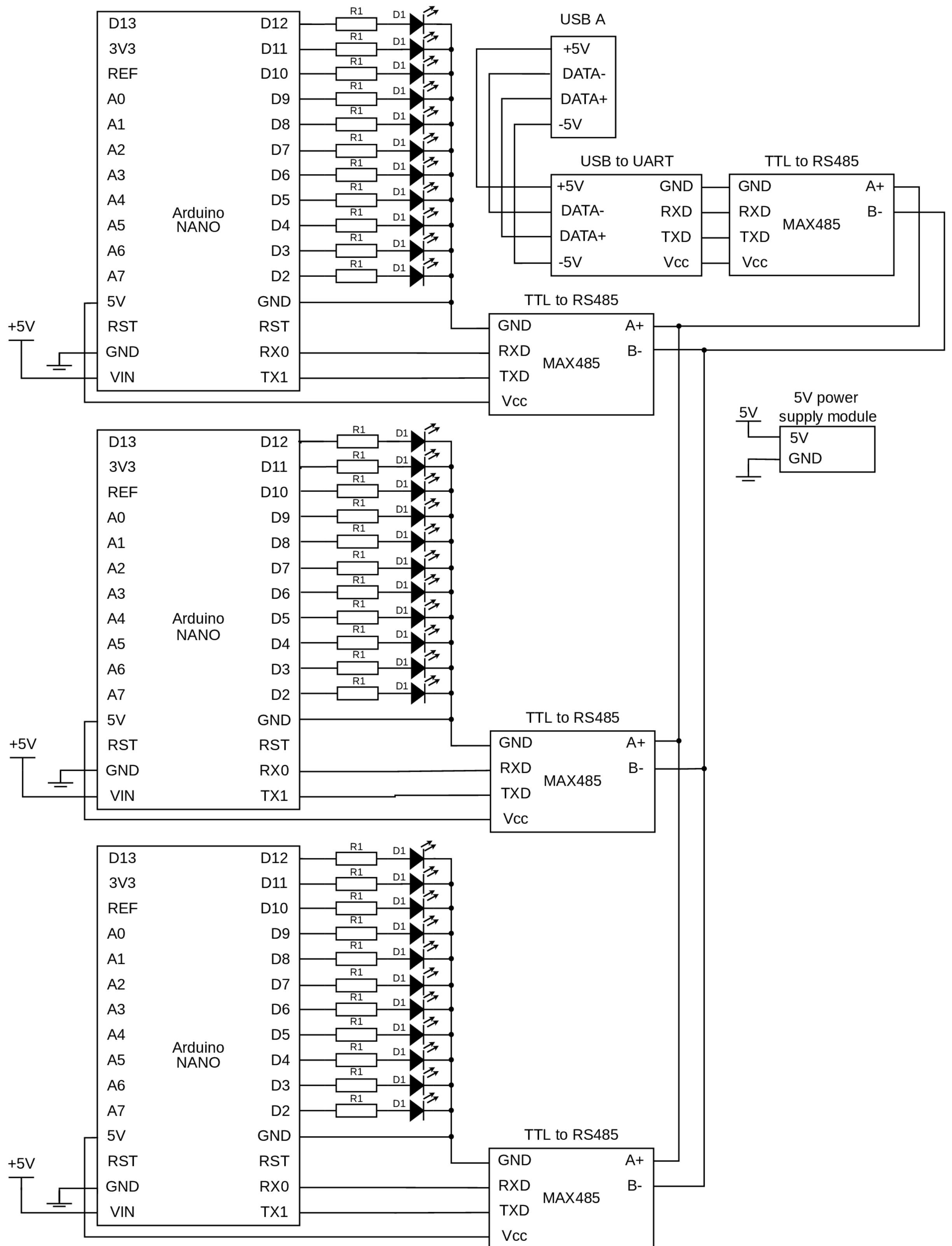

Fig. 5.1: Schematic for Demonstration unit

Conclusion

The preliminary research was conducted in preparation for the development of a Forth library implementing MODBUS RTU slave behaviour. Both the Modbus protocol and the Forth language were researched and are described as part of the theoretical portion of the thesis.

The MODBUS protocol is described in the first chapter of this document, focusing on the RTU variant implemented in the library.

The description of the basics of the Forth programming language lies in the second chapter. The Starting FORTH [3] book was an excellent introduction to both the programming in the Forth language as well as Forth system inner workings.

A selection of Forth compilers are listed and described in the third chapter. For the development of the library itself, FlashForth was chosen. It differs from the Forth language standard, mostly in order to provide a more robust system that is less susceptible to user errors. Some of the differences proved to be challenges during the development, such as the missing DO ... LOOP cycle FlashForth does not implement by default. All of these issues were addressed over the course of the development and none of them held up the creation of the library for too long. Furthermore the robust nature of FlashForth proved quite useful, especially in the early stages of the development when I was still acquainting myself with Forth programming in general as multiple erroneous operations were prevented, such as writes to FlashForth kernel by passing the address and value to memory write words in the wrong order.

The library itself was successfully developed. The development brought challenges, many of which are described in the last to final chapter of the thesis. Both the inner workings of the library itself as well as the process of its creation are described in this chapter. The resulting library is a part of the digital appendix A.

A demonstration unit was constructed using the library in order to prove the library is functioning as intended. Three Arduino NANO boards running a slightly modified library were used as part of the unit. The code for the units is also included in the digital appendix A.

Bibliography

[1] Modbus organization: *MODBUS APPLICATION PROTOCOL SPECIFICATION V1.1b3* [online]. 2012 [cit. 19. 10. 2021]. Available from URL: <https://modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf>.

[2] Modbus organization: *MODBUS over Serial Line Specification and Implementation Guide V1.02* [online]. 2006 [cit. 15. 11. 2021]. Available from URL: <https://modbus.org/docs/Modbus_over_serial_line_V1_02.pdf>.

[3] BRODIE, L.: *Starting FORTH:an introduction to the FORTH language and operating system for beginners and professionals* [online]. Englewood Cliffs. N.J.: Prentice-Hall, ©1981 [cit. 11. 10. 2021]. Available from URL: <https://www.forth.com/wp-content/uploads/2018/01/Starting-FORTH.pdf>.

[4] WÄLDE, E.: *Nodes on a RS485 Bus* [online]. 2015 [cit. 18. 11. 2021]. Available from URL: <http://amforth.sourceforge.net/Projects/RS485/RS485Bus.html#id1>.

[5] [NORDMAN, M.]: *FlashForth Guide* [online]. [cca 2015] [cit. 12. 11. 2021]. Available from URL: <https://flashforth.com/index.html>.

[6] TING, C.H. : *Tao of Arduino* [online]. 2011 [cit. 30. 12. 2021]. Available from URL: <https://gitlab.com/jjonethal/eforth328/-/blob/master/ForthArduino_1.pdf>.

[7] Schneider Electric: *Modicon is now Schneider Electric* [online]. [cit. 20. 11. 2021]. Available from URL: <https://www.se.com/uk/en/about-us/company-profile/brands/modicon.jsp>.

[8] HMS: *Industrial network market shares 2020 according to HMS Networks* [online]. May 29, 2020 [cit. 20. 11. 2021]. Available from URL: <https://www.hms-networks.com/news-and-insights/news-from-hms/2020/05/29/industrial-network-market-shares-2020-according-to-hms-networks>.

[9] MPE Microprocessor Engineering: *Comet Landing – a triumph for Forth in Hardware and Forth in Software* [online]. November 13, 2014 [cit. 6. 12. 2021].

Available from URL:
<http://www.mpeforth.com/press/MPE_PR_From_Telescope_to_Comet_2014_11_13.pdf>.

[10] FORTH, Inc.: *Forth in Space Applications* [online]. ©2022 [cit. 3. 1. 2022]. Available from URL:
<https://www.forth.com/resources/space-applications/>.

[11] Forth Interest Group: *Forth Compilers Page* [online]. [cca 2013] [cit. 12. 12. 2021]. Available from URL:
<http://www.forth.org/compilers.html>.

[12] *AmForth Technical Guide* [online]. [cca 2014] [cit. 29. 12. 2021]. Available from URL:
<http://amforth.sourceforge.net/TG/TG.html>.

[13] *ESP32forth website* [online]. [cit. 29. 12. 2021]. Available from URL:
<https://esp32forth.appspot.com/ESP32forth.html>.

[14] Atmel: *ATmega328p DATASHEET* [online]. January ,2015 [cit. 17. 5. 2022]. Available from URL:
<https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf>.

List of appendices

A Contents of the digital appendix **47**

A Contents of the digital appendix

The attached digital medium contains the following files:

```
/ ...................................... root folder of attached medium
├── library ................................... library source files
│   ├── config.fs ............ memory mapping and device configuration
│   ├── Atmega328p.fs ................... words specific to Atmega328p
│   ├── library.fs ................................ device-agnostic words
│   └── debug.fs ............................ words useful for debugging
├── application ............... source files for the demonstration unit
│   ├── library.fs ............. modified library with injected mapping
│   └── application.fs ................................ application words
├── FlashForth
│   └── ff.hex ............ FlashForth image compiled using XC8 in MPLAB
└── Failed_State_machine
    └── state_machine.fs ......... scrapped implementation of Modbus server
```

The library folder contains all the files constituting the finished library itself, ready to be interpreted on an Atmega328p. The files used for the demonstration unit are placed in the application folder. A FlashForth image with CPU load measurement disabled that was used in the development is present in the FlashForth folder. It was compiled using the XC8 v2.36 compiler in MPLAB v6.00 from the FlashForth v5 source files. The Failed_State_Machine folder contains a Forth source file implementing the state machine design that was ultimately aborted during the development.

In the physical rendition of the thesis, the digital appendix is contained on a CD along with the digital version of this thesis as well.