A usable, real-time interpreter for rapid product development

by

Johannes Jacobus Jordaan

A dissertation submitted to the Faculty of Engineering

in partial fulfilment of the requirement for the degree

MASTER OF ENGINEERING

in

COMPUTER AND ELECTRONIC ENGINEERING

at the

NORTH-WEST UNIVERSITY – POTCHEFSTROOM CAMPUS

Supervisor: Prof JEW Holm

May 2010

ACKNOWLEDGEMENTS

I want to express my gratitude to the following people for their support and guidance, without whom this project would not have been possible.

Firstly, I thank my supervisor, Prof. Johann Holm, for his expert guidance, support and motivation throughout and beyond the duration of the project.

I thank my parents for their continuous love and support and for paving my way which led to this opportunity.

I thank my family and friends for their inputs, support and motivation.

Above all else, I thank God for providing me with the opportunity and ability to complete this project.

SOLI DEO GLORIA

"For the Glory of God Alone"

ABSTRACT

The need for a usable programmable logic controller (PLC) was identified. This need originated from the shortfalls in flexibility and limited functionality of existing industrial-type PLCs. In order to address these shortfalls, a flexible low-level PLC core is proposed as an alternative. This low-level system consists of basic instructions that can be chained to perform any high-level PLC function. The basic instructions also support additional functionality that lack in traditional PLCs.

The low-level system is built around a *virtual machine* that implements a low-level instruction set. The virtual machine is implemented on the firmware of the PLC - this is done to provide an abstraction layer at a relatively high level so that the machine is portable.

An *embedded operating system* is implemented on the PLC to provide an additional, lower-level abstraction layer between the virtual machine and PLC hardware. The selection of a suitable operating system required an experiment with which to compare operating system performances. It was found that uCLinux is not suitable for deterministic real-time applications and that FreeRTOS performs better.

The PLC is programmed and configured through the use of a *specialised computer application*. The programmed PLC applications are structured in *event-action* pairs that specify which actions to take when an event occurred. To improve the usability of the system, the actual programming is done in an intuitive, human-understandable programming language based on *if-then-else clauses*.

All input requirements were addressed by means of design. Furthermore, design guidelines and a design philosophy were derived from an analysis of tried and tested PLC principles. The verification of the functional capability of the interpreter was demonstrated in the operating system comparison experiment. This shows that the requirements for this project were addressed in a successful interpreter design.

OPSOMMING

Die behoefte aan ‘n bruikbare, programmeerbare logiese beheerder (PLC) was geïdentifiseer. Hierdie behoefte se oorsprong lê gesetel in die tekortkominge van bestaande industriële PLCs in terme van hul beperkte funksionaliteit. Om hierdie tekortkominge aan te spreek, word ‘n buigbare, lae-vlak PLC kern as alternatief voorgestel. Hierdie lae-vlak stelsel bestaan uit basiese instruksies wat gekombineer word om enige hoë-vlak PLC funksie te skep. Hierdie basiese instruksies steun ook addisionele funksionaliteit wat ontbreek in tradisionele PLCs.

Die lae-vlak stelsel is gebasseer op ‘n *virtuele masjien* wat die lae-vlak instruksiestel implementeer. Die virtuele masjien is geïmplementeer op die ingebedde sagteware van die PLC – dit bied ‘n abstraksielaag op ‘n relatiewe hoë vlak sodat die masjien oordraagbaar is.

‘n *Ingebedde bedryfstelsel* is geïmplementeer op die PLC, om ‘n addisionele, lae-vlak abstraksielaag tussen die virtuele masjien en die PLC hardeware te lewer. Die seleksie van ‘n geskikte bedryfstelsel het ‘n eksperiment vereis, om die dienslewering van bedryfstelsels te evalueer. Daar is gevind dat uCLinux nie geskik vir deterministiese intydse toepassings is nie en dat FreeFRTOS beter diens lewer.

Die PLC word geprogrammeer en gekonfigureer deur van *gespesialiseerde rekenaar sagteware* gebruik te maak. Die PLC toepassings is gestruktureer in *gebeurtenis-aksie pare* wat die aksies spesifiseer wat moet volg op ‘n spesifieke gebeurtenis. Die programmering word gedoen deur gebruik te maak van intuïtiewe, mensverstaanbare programmeertaal wat gebasseer is op “*if-then-else*” klousules.

Al die insetvereistes was geadresseer deur middel van ontwerp. Verder was die ontwerpsriglyne en –filosofie afgelei uit ‘n analise van bewese PLC beginsels. Die verifikasie van die funksionele bevoegdheid is gedemonstreer in die bedryfstelsel eksperiment. Dit toon aan dat die vereistes van die projek suksesvol aangespreek word deur ‘n suksesvolle interpreteerder ontwerp.

TABLE OF CONTENTS

TABLES AND FIGURES

ABBREVIATIONS AND ACRONYMS

CPU – Central Processing Unit

FPGA – Field Programmable Gate Array

GSM – Global System for Mobile Communications

HAL – Hardware Abstraction Layer

IC – Integrated Circuit

I/O – Input or Output

LCD – Liquid Crystal Display

LE – Logic Element

OS – Operating System

PC – Personal Computer

RAM – Random Access Memory

ROM – Read Only Memory

RTOS – Real-Time Operating System

SIM – Subscriber Identity Module

TCP/IP – Transfer Control Protocol / Internet Protocol

VM – Virtual Machine

CHAPTER 1
INTRODUCTION

1.1 PROJECT OVERVIEW

A client identified the need for a usable programmable logic controller (PLC). The PLC is to be programmed by a PLC programmer using a computer and specialised PLC development software. What differentiates this PLC from existing PLCs is the specific requirement for usability, which dictates that the device must be easily programmable and robust. Since a product with these requirements was not found to exist, a development project was defined to bring such a PLC system into being.

The system essentially comprises a programmer (person), specialised computer software, a PLC and a back-end system that provides a communications link between the PLC and other systems. The computer software can establish a communication channel to the PLC in order to download new user programmes and monitor the status of the PLC. The downloaded user programs are also interpreted and executed on the PLC in event-action pairs by using an interpreter.

The focus of this specific development is thus the design and development of an interpreter that interprets and executes event-action pairs. This must be done with focus placed on usability, portability, and robustness of the system. In order to achieve this, it was necessary to research and understand existing PLC solutions, operating systems, interpreters (scripts and scripting engines), hardware abstraction, and hardware implementation strategies.

As an alternative to the development of an interpreter, it could have been possible to build an interface between existing PLC frameworks (for example, ladder logic) and a user interface. However, additional requirements such as the need for control over software code, flexibility, and portability necessitated the development of an interpreter that can be adapted to suit the purposes of the client.

It is important to note here that the design and verification of the work in this research and development project, was done on digital and analog inputs and digital outputs in order to demonstrate the functional capability of the interpreter of the PLC. Additional functions can be demonstrated in exactly the same fashion as this input / output set and, although not shown here, the additional functions were actually implemented. Therefore, additional PLC features are not described in this document for the sake of readability and without any material effect on the comprehensiveness of this work – the focus in this document is on the design and implementation of the interpreter.

1.2 BACKGROUND AND MOTIVATION

1.2.1 SUB-SYSTEM OVERVIEW

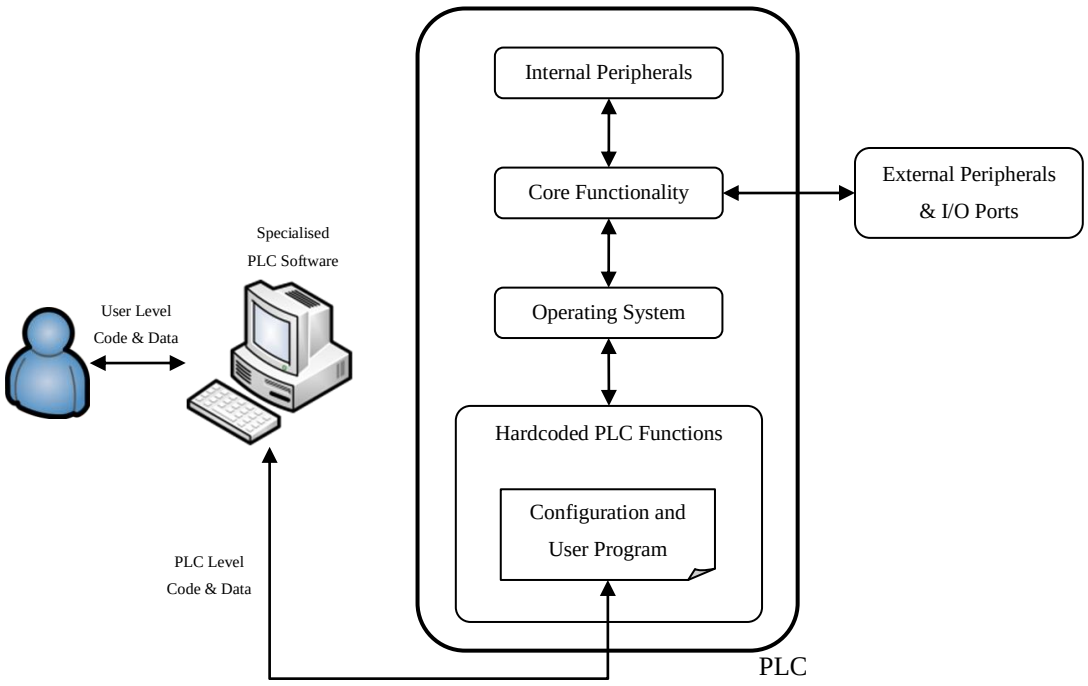


FIGURE 1-1: HIGH-LEVEL OVERVIEW OF EXISTING PLC

A high level overview of a PLC device is shown in Figure 1-1. It consists of analog input ports and digital input and output ports, as well as standard communication ports and a GSM module. The PLC device is programmed with the use of a computer and PLC development software (similar to software that supports ladder logic or the like).

In PLCs, user applications are based on and limited to event-based actions. For example, an event described in English (not ladder logic) could be: "if digital input 1 transitions from low to high then set digital output 5 to high." These user-level instructions are translated to PLC level configuration data by the PLC computer software and are then programmed onto the PLC device. Therefore, through the PLC development software, the user has access to higher-level "soft" logical functions that essentially provide a logical programming construct to lower level hard-coded functions.

The types of events (for example: transition on a digital input) and types of actions (for example: change digital output) are usually hard coded in the firmware of PLC devices. The user can select events and actions from this predetermined list of events and actions and can chain several of these to form an application.

1.2.2 SHORTFALLS OF EXISTING PLC SYSTEMS

The primary shortfall with the existing hardcoded functionality is that the flexibility available to the user is limited to "traditional" industrial PLC events and actions. There is, however, a *need for more general types of events and actions that users can easily understand*. This includes support for data types such as multi-media.

A further shortfall concerns the *difficulty of use of PLC programming software* and the actual programming of the PLC itself. Existing PLC programming software limits the scope of users to only experts, which is partially due to the complexity of learning and understanding ladder logic. On the surface this limitation appears to lie mainly with the PC software and not with the low-level (PLC) firmware. However, it is necessary to know that the high-level software functionality is directly dependent on the interfaces provided by the low-level (PLC) firmware. This is because the PC software cannot provide any functionality that does not exist in the PLC firmware definition. Therefore, the shortfall of the high-level software is also affected by the limitations of the low-level firmware.

This shows that the primary shortfall lies with the definition of the existing PLC constructs and that the events and actions are limited to provide mostly logical functions.

Finally, and importantly, the need to contain any operational *metadata on the PLC* is not addressed by existing PLCs. In other words, the PLC does not keep a record of the complete configuration of the operational application, but only of the configuration pertaining to the PLC's specific configuration.

1.2.3 PROPOSED ALTERNATIVE

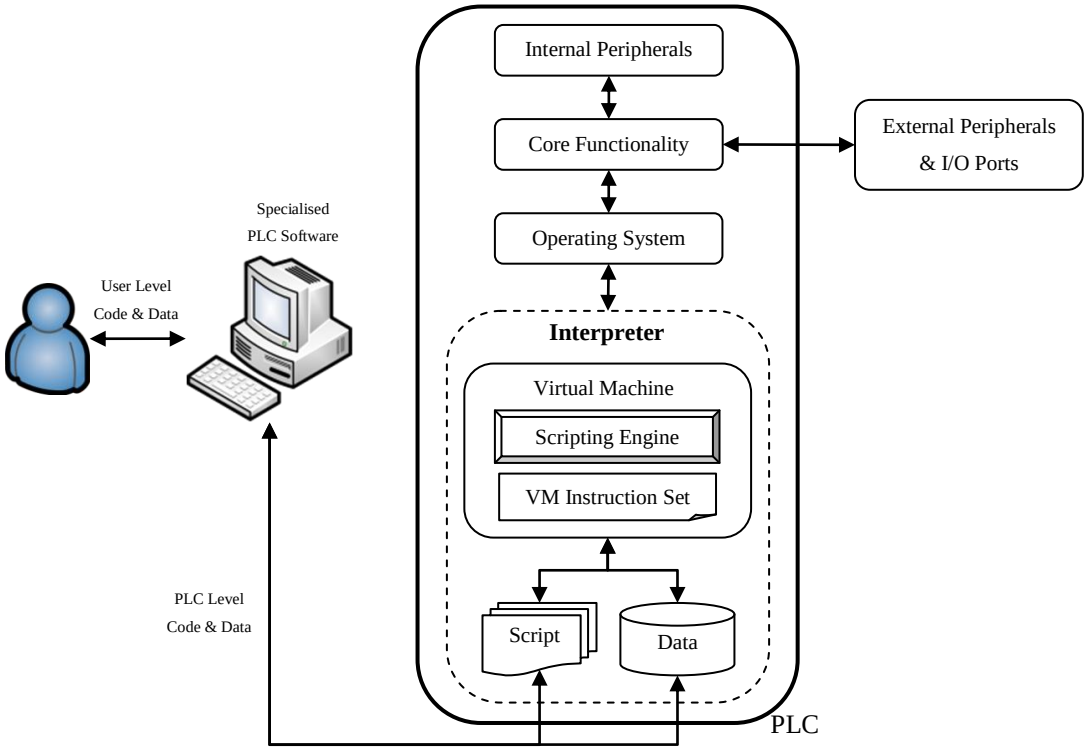


FIGURE 1-2: PROPOSED ALTERNATIVE

The proposed alternative, to address the limited functionality of existing PLC systems, is to replace the hard-coded PLC firmware section (see Figure 1-1) with a flexible low-level system. That is, a set of low-level operations that can be chained to perform any high-level operation (previously limited to a reduced set of PLC functions). The principle behind this approach is to implement a *virtual machine* (with its own unique instruction set) that runs within the firmware of the actual hardware platform. This virtual machine (VM) can then execute applications (written with the VM instruction set) as if it were a physical processor, also with access to the PLC's hardware functionality.

At this point the VM might seem redundant, seeing that the PLC already has its own processor with which to execute applications. The difference, however, is that the VM is only responsible for executing applications that are programmed by the user. This is a small fraction of responsibility compared to the PLC firmware that needs to manage all hardware configurations and servicing of interrupts, peripherals, and inputs and outputs (I/O). Furthermore, if the user application was to be executed by the PLC's processor directly, it would require a complete recompilation of all the firmware.

To improve robustness of the system, a proven operating system is inserted between the VM and core functionality of the hardware architecture. The operating system manages all hardware specific tasks, while providing the VM with a stable and reliable interface to access underlying hardware functionality. This abstraction layer provided by the operating system, together with the VM and VM instruction set, greatly improves the portability of the system.

The virtual machine is discussed in more detail below and is shown again for convenience:

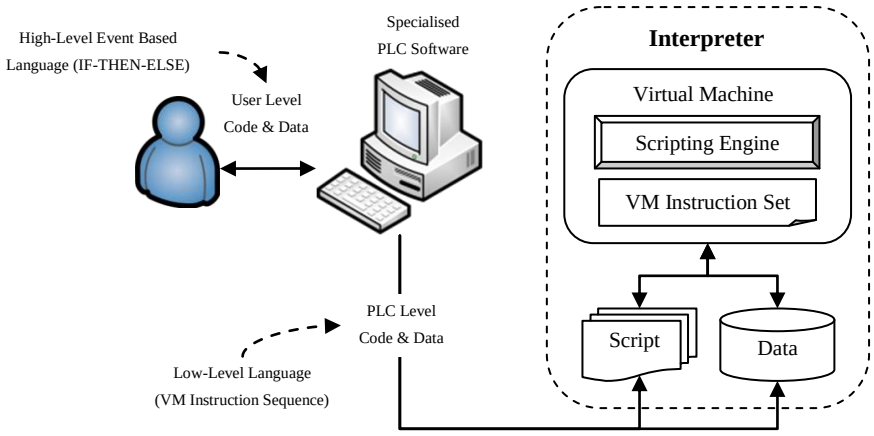


FIGURE 1-3: VIRTUAL MACHINE

The virtual machine consists of a scripting engine that interprets (executes) applications that are programmed in the form of a script, where a script is a sequential list of operations or instructions that tell the scripting engine what action to perform. (Note that this is equivalent to a program consisting of machine instructions that is executed by a hardware processor). The collection of all available script operations form the *VM instruction set* and the complete set of instructions is listed in Appendix C.

The entire concept of a virtual machine, scripting engine, script, and internal data is referred to as the interpreter and forms the focus of this dissertation.

The programming of the interpreter (specifically the scripts) is done by a programmer using *specialised PLC computer software*. This PC software is focused on providing the PLC programmer with an easy-to-use and user-friendly working environment. The key to achieving this is to simplify the programming language to the extent that it is intuitively understandable to humans. This led to an event-based programming language based on logical *if-then-else* clauses. This simplistically means that every event starts by defining a condition (if) which, when satisfied, will cause the PLC to perform an action (then). If the condition is not satisfied, an optional alternative action can be specified (else). For example, *if* digital input 1 is high *then* set digital output 5 to high *else* set digital output 5 to low.

These programmed events are converted (compiled) to a *script consisting of VM instructions* that can be executed by the PLC (more specifically, the interpreter). The actual transfer of the script from the PC to the PLC is done using a host based command protocol, sent over a communication channel between them. A complete list of these *computer ⇔ PLC commands* is listed in Appendix B.

1.3 PROJECT OBJECTIVES

The high level project objectives are stated in summarized fashion, as described below.

1.3.1 PRIMARY OBJECTIVE

The primary objective is to design and program a real-time interpreter system for a PLC.

1.3.2 SECONDARY OBJECTIVES

In order to achieve the primary objective, the following general system elements must be addressed:

- A *computer interface* that is easy to use with rule-based logic functions and configuration options;
- A reliable *communication protocol* to interface the PLC to the computer software;
- An *interpreter* to execute all user-programmed applications on the PLC platform;
- An *embedded operating system* that is recognized, stable, and that provides real-time functionality;
- A functional *PLC module* that integrates all elements of the system.

1.4 PROJECT WORK

A research and development project was defined to realize the PLC system. The work done in this project includes the following:

- Perform a literature study on existing PLCs, embedded systems, programming trends (from assembly language to scripting language), and operating systems. Although some of the work may seem well-known, it was necessary to do this study in order to review and *understand the philosophies behind these tried and tested technologies* (chapter 2);
- Design and program the specialised PLC computer application that supports the requirements as set out above. (*Note that the graphical user interface for the computer application did not form part of the actual research and was done in order to support the development of an interpreter. Thus, the graphical user interface design is not included in this document)*;
- Design and implement a high-level compiler (from source code in the form of *if-then-else* clauses to virtual machine code (VM instruction set sequence) (chapters 3 and 4);

- Identify and implement viable operating system alternatives on the PLC hardware so that a definitive selection of the best operating system (for this project) can be made based on comparative performance tests (chapters 3 to 5);
- Perform an experiment to determine which operating system provides the best performance in terms of execution speed and real-time capabilities. This was necessary since a direct and quantitative comparison between suitable operating systems did not exist at the onset of this research (chapter 5);
- Design and implement a computer ⇔ PLC protocol for communication and programming of the PLC (chapters 3 and 4);
- Design and implement a virtual machine and its complete instruction set (chapters 3 and 4);
- Integrate the interpreter on operating system options in order to test and evaluate both the interpreter and the operating system options (chapters 3 and 4).

1.5 SPECIFIC REQUIREMENTS

The input requirements for the project require *specific functionality and characteristics* to be implemented, as listed below:

- GSM-communication (SMS, GPRS);
- Analog signal measurement;
- Digital input measurement;
- Digital output manipulation;
- Pulse counting;
- Event timing;
- Mathematical manipulation of data;
- Programmer definable variables for data handling;
- Windows-based PLC configuration and programming;
- Intuitive and logical programming language and interface;
- General manipulation of data and outputs using rule-based events;
- Connectivity between PLC and computer via RS-232, USB, or GPRS;
- Remote monitoring of PLC's I/O and data;
- Portable and robust code on the PLC module itself.

The aforementioned functionality must be implemented by taking *general design requirements* into account. The design requirements are derived from the aforementioned *input requirements* (high-level), as well as requirements that result from a *technology* and *literature study* (as performed in Chapter 2). These design requirements manifest themselves as design *constraints* and *guidelines*, as follows:

PORTABILITY

In order to achieve portability, the following must be used:

- An abstraction layer must be created by using a recognized operating system;
- A virtual machine should be created with which to allow multi-platform operation;
- A generic scripting language with interpreter should run on top of the virtual machine.

USABILITY

In order to make the product usable, the following should be done:

- Create user-friendly computer software (easy to use, intuitive design);
- Use flexible event-based programming at high level;
- Provide feature to evaluate events at specific times or fixed periodic intervals;
- Use an intuitive human-understandable programming language (if-then-else clauses).

ROBUSTNESS

The interpreter should be robust when the following is taken into account:

- A stable, recognized operating system must provide a reliable interface between hardware and firmware;
- A virtual machine will provide additional robustness in that the machine allows only recognized instructions that do not have direct access to low-level functionality;
- A single code path should be used (sequential execution);
- Events must be independent of other events at the lowest level – high level logic may link events so that the user can construct their own logic structure;
- Inputs are synchronized by only reading input ports once at the onset of a program cycle. External changes to inputs during the event evaluation period are recorded but only evaluated at the start of a next cycle – this means that a deterministic, synchronous system is required;
- Changes to output ports only take effect at the end of each program cycle – in effect also synchronizing outputs.

All the aforementioned requirements, both high-level input requirements and derived design requirements, shall be addressed in the analysis and design of this work.

1.6 OVERVIEW OF THIS DOCUMENT

Chapter 2 summarizes all relevant research done for this project. This includes information on PLCs, embedded systems, programming trends (from assembly language to scripting language), operating systems, and generic product development aspects.

Chapter 3 focuses on the preliminary design of an interpreter system for a PLC. A functional analysis is performed on a typical PLC project which leads to the functional architecture for this project. Both these sections are explained with detailed descriptions. As a final contribution, the resource allocation is provided in table form providing the link between function and architecture.

Chapter 4 gives the detailed design of the project elements, which include the design and programming of the PLC computer software *(including high-level programming language and compiler but excluding GUI)* and the design and programming of the PLC firmware *(including virtual machine, VM instruction set, interpreter, and scripts)*.

Chapter 5 gives account of the comparative experiment that was performed to determine the most suitable operating system for this system.

The appendix provides detailed lists for:

- PLC programming interfaces *(for example, digital input or digital output)*;
- Computer ⇔ PLC commands *(commands for controlling and programming the PLC from the computer software)*;
- Virtual machine instruction set;
- System calls *(the mechanism through which the virtual machine can access the PLC hardware and system functionality).*

1.7 SUMMARY

In summary, existing PLCs lack functionality and ease of use. Therefore, an alternative type of PLC was proposed that addresses the shortfalls of existing PLCs. The proposed PLC provides flexible low-level instructions that are chained to form an extended high-level program. The programming environment is available to the PLC programmer through an intuitively human understandable, event-based language (if-then-else clauses).

The design and implementation of the graphical user interface is not discussed in this document, although it is a necessary component of the overall PLC system. Hardware design, although necessary, did not form part of this research and is specifically excluded from this work.

The PLC that is proposed in this work was designed and implemented successfully. This document provides the technical detail of the compiler and interpreter design as well as the design decisions that were taken in order to realize a *usable, real-time interpreter*.

Account is also given of an experiment that was done to assess the performance of different operating system alternatives.

CHAPTER 2
LITERATURE STUDY

2.1 INTRODUCTION

This chapter provides a theoretical background that relates to the development of an embedded solution. For example, development typically starts with an operational-level process that is systematically broken down to the lowest levels of firmware and hardware development.

The aspects of this process are explained and organized in the following order:

- Overview of a typical product life cycle;
- Available hardware technology (processing technologies);
- Evolution of embedded software development and current programming trends;
- Characteristics of operating systems;
- Available embedded operating system alternatives;
- Embedded development tools with focus on open source alternatives;
- Supporting aspects of embedded projects.

2.2 PRODUCT LIFE CYCLE

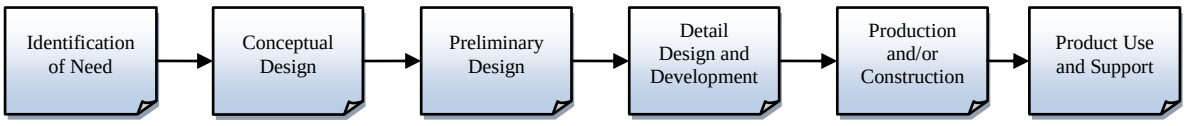


FIGURE 2-1: PRODUCT LIFE CYCLE

A system or product starts its life cycle with the identification of a problem or need. This leads to the conceptual design that focuses more on *what* needs to be done, rather than on the details of *how* it is to be implemented. The output of this phase is usually a concept specification that provides the operational definition of the system [1].

After the conceptual design, a preliminary design phase reduces project and product risk by analyzing the operational requirements in further detail. The result of this analysis is usually a development specification that defines the detail design phase [1].

The next phase is detail design and entails all aspects and details of the final product down to activities such as component selection, circuit design, and software development. It typically includes several iterations of design, implementation, and testing. Due to the high level of detail involved and the vast number of factors to consider, this phase also consumes the bulk of the development time, work, and cost [1].

It is for the above reason that a high-level interpreter must be used. That is, in order to reduce the detail design activities, it is necessary to provide a platform on which detail work of a project has been addressed and fully tested. This allows the designer to spend more time on the preliminary design, which reduces both risk and time to market.

Once the product's detail design and testing have been completed, it enters production and is ultimately used in its intended application area [1].

2.3 EMBEDDED HARDWARE

It is necessary to consider different hardware technologies, as possible options for an interpreter platform. Although most of these technologies are well known, a basic overview is given for the sake of completeness and also serves to revisit the design philosophy of processing architectures (on which to base the VM design).

2.3.1 MICROCONTROLLER

A microcontroller is a single integrated circuit (IC) that consists of a central processing unit (CPU) and most of the peripherals required by a microprocessor system. These peripherals typically include, but are not limited to [2]:

- Random access memory (RAM);
- Read only memory (ROM);
- Interrupt controller;
- Serial interface;
- Timers.

An internal communication bus connects these peripherals as shown in Figure 2-2.

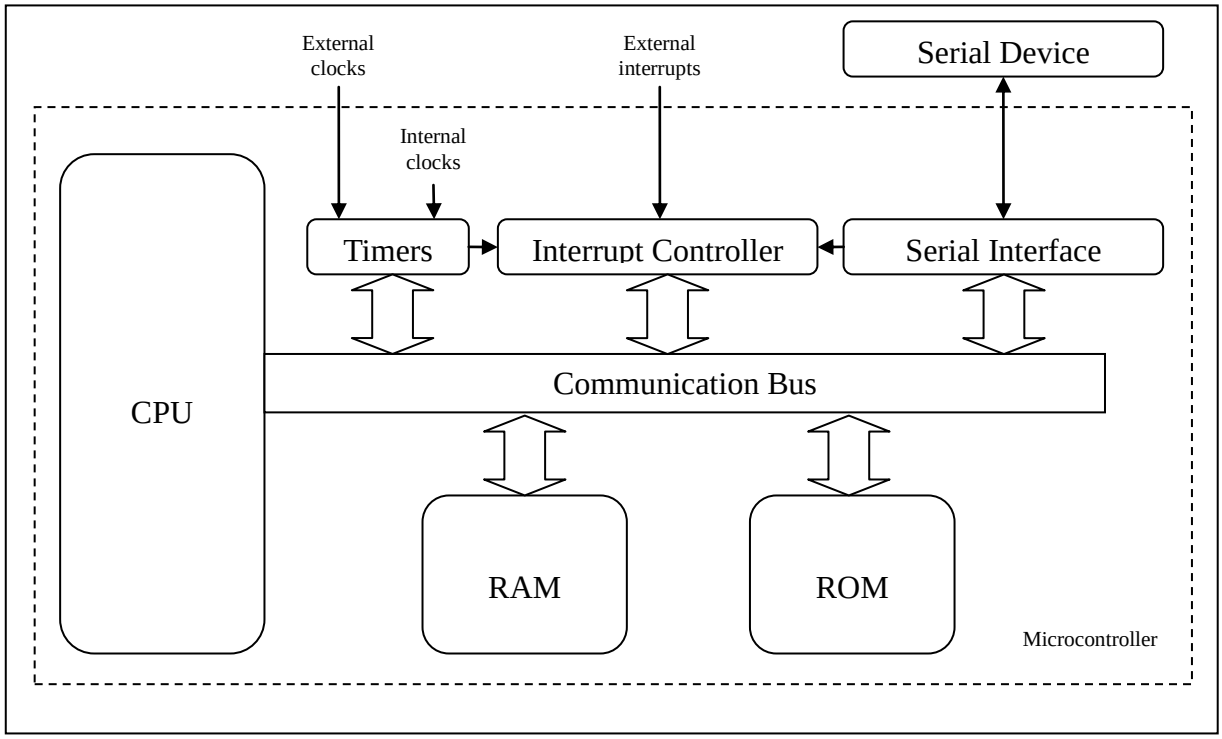


FIGURE 2-2: INTERNAL ARCHITECTURE OF A MICROCONTROLLER

Microcontrollers are intended for industrial or consumer products, typically where a form of control is required. The user of these products is usually unaware of the existence of an embedded controller. For example, the user is (usually) unaware of the microprocessor in a digitally controlled microwave oven or automobile anti-lock braking systems [2].

2.3.2 FIELD PROGRAMMABLE GATE ARRAY

A field programmable gate array (FPGA) is an integrated circuit from the field programmable device[1] (FPD) family. As all FPDs, FPGAs are designed for applications where logic elements or digital hardware is required. This solution is also flexible and not platform specific (for example, the FPGA can use a few elementary logic functions or emulate an entire microcontroller in a complex combination of logical units) [3][4].

In the family of FPDs, FPGAs offer a very high logic density as well as a high ratio of flip-flops[2] to logic resources [3][4].

[1] A *field programmable device*, or FPD, is an integrated circuit capable of being programmed by a user, to implement various digital hardware functionality [3].

[2] A *flip-flop* is a bistable electronic circuit, capable of storing a single bit (either 0 or 1). Its output is dependent on the value stored (high for 1 and low for 0) [4].

The basic internal structure of an FPGA is shown in Figure 2-3.

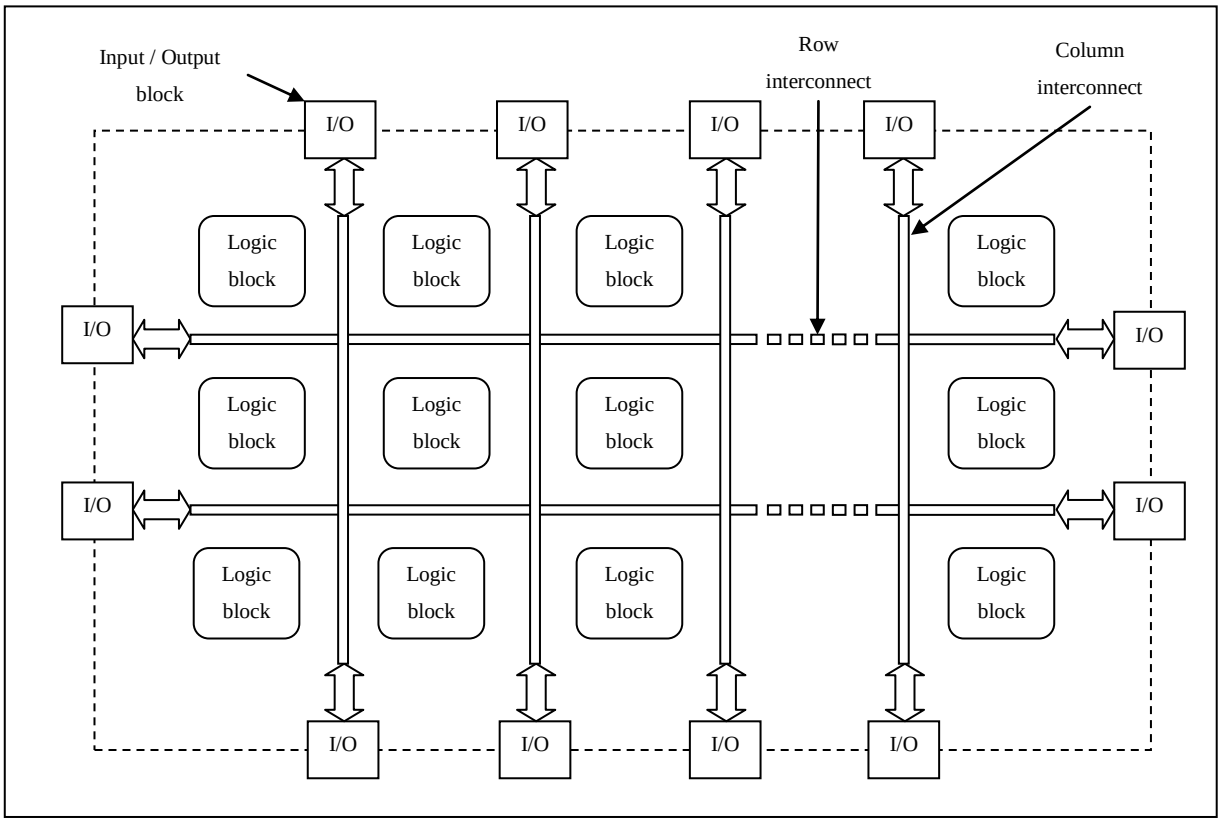


FIGURE 2-3: FPGA STRUCTURE

The FPGA structure is based on the concept of a logic element (LE). Each logic element is capable of implementing a programmable logic function in conjunction with an optional (selectable through programming) flip-flop. These flip-flops can be used to store the output of a logic element, thus providing the FPGA with a form of memory. Several logic elements are grouped to form logic blocks. These logic blocks, together with I/O blocks, are cascaded in an array, with programmable connections between them called interconnects [3][4].

2.3.3 PROGRAMMABLE LOGIC CONTROLLER

A programmable logic controller (PLC) is a powerful tool and currently the leading technology in manufacturing automation. Before PLCs, automation was controlled by hardwired relay panels. This technology had distinct problems, which directly led to the development of PLCs [5].

The first PLCs addressed the problems with relay systems, but did not provide any additional functionality. Later versions, however, improved on these basic features and currently several categories of PLCs are available. Table 1 provides a summary of these categories [5].

TABLE 1: PLC CATEGORY AND FEATURES

PLC Category	I/O Points	Program memory (Words)	Added features
Micro	32	2K	• Basic relay instructions • Timers • Counters
Small	128	4K	• Analog I/O • Shift registers • Sequencer instructions • Primitive communication with other PLCs
Medium	1024	32K	• Remote I/O • Basic math instructions • Data handling instructions • Subroutines • Interrupts • Functional block or high-level language • Local area network connection
Large	2048	256K	• Enhanced math instructions • Enhanced data handling instructions • PID control
Very Large	8192	4M	

Because PLCs replaced relay panels, the language developed for programming PLCs was based on symbols used in relay diagrams. This new programming language was called ladder logic and is still the most popular language amongst PLC programmers [5].

To illustrate the fundamentals of ladder logic, a few of the programming symbols must first be introduced. These are shown in Table 2.

TABLE 2: LADDER LOGIC SYMBOLS

Name	Symbol	Description
Normally open contact	Input 	Allows current to flow through if the input is ON (for example the button is pressed)
Normally closed contact	Input 	Allows current to flow through if the input is OFF (for example the button is not pressed)
Output	Output 	Sets output to ON if current reaches this point, otherwise output is OFF

These symbols are chained, in a ladder-like structure, to form a PLC application. An application is scanned (executed) from top (start) to bottom (end) from left to right and then repeated from the top [5]. This ensures that program flow takes place in a fixed and deterministic fashion. The same principle, of using a *fixed sequence of events*, was used in the development of the interpreter described in this thesis.

Execution can be understood by thinking in terms of current flow. Current starts to flow at the left side of each branch. When the current reaches an open contact (normally open contact in the off state, or normally closed contact in the on state), the flow is broken and cannot continue along that path. However, if current reaches a closed contact (normally open contact in the on state, or normally closed contact in the off state), current flow will continue. While there is a path for current to flow through an output, that output will be on. Otherwise that output will be in an off state.

An important aspect of PLC applications is that all *input states are read into memory* at the top of the application, that is, at the *start of the sequence*. Only these states in memory are used during the application and they remain unchanged till the end of the sequence. Thus, any change of input states during a cycle from top to bottom, will have no effect until the start of the next cycle. Similarly, any changes to output states are modified in memory and written out only at the end of a program cycle. This improves application reliability and forces the system to always operate in a deterministic way [5]. This form of *I/O-synchronization* was applied in the development of the interpreter in this thesis, in order to provide a deterministic and synchronized system.

Figure 2-4 shows an example ladder logic application, followed by a description of how it works.

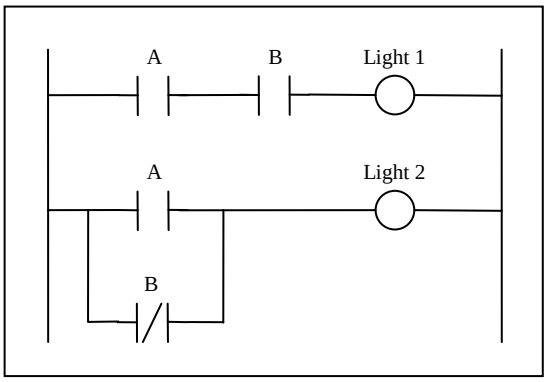


FIGURE 2-4: LADDER LOGIC EXAMPLE APPLICATION

For this example, consider inputs A and B to be buttons and the outputs to represent two lights. Execution starts from the top and at the first branch. This branch contains two normally open contacts, followed by the output to Light 1. A normally open contact only lets current through if it is closed; for this example while a button is pressed. Therefore, Light 1 will only be on while both buttons A and B are pressed.

The second branch contains a normally closed contact. By following the same logic as before, the normally closed contact will only allow current through if the button is not pressed. Because the two contacts are in parallel, there are two paths for current to reach the light. Therefore, Light 2 will only be on while either button A is pressed or button B is not pressed.

2.4 EMBEDDED SOFTWARE

Irrespective of hardware, application software is required for an embedded device to fulfill its purpose [4]. Embedded application software is referred to as firmware [2]. Computer programming, along with firmware programming, has undergone vast evolutionary changes over the last century [6].

2.4.1 MACHINE LANGUAGE

All microprocessors operate by executing a program comprised of a sequence of instructions. Each of these instructions is represented by a binary sequence and is processor specific. The set of binary sequences is defined by the designers of the processor and referred to as the machine language [4].

Inputting these binary sequences directly was the first form of processor programming and is still possible today. This, however, is very difficult and prone to error. Due to the complexity of working only with binary sequences, assembly language was introduced [4].

2.4.2 ASSEMBLY LANGUAGE

Assembly language replaces the binary sequences with easy to remember alphabetic mnemonics. Each mnemonic represents a specific binary sequence and is known as an op-code. Some op-codes take one or two operands, each representing either a constant, register or memory location. Assembly language programs are assembled into machine code by a program called an assembler [4].

Because assembly op-codes have a one to one relationship to its corresponding binary sequence, assembly language programs are processor specific and not portable. The programmer must also consider all aspects of the program, including register allocation and function call transitions [4][6].

2.4.3 SYSTEM LANGUAGES

From the late 1950s several higher level languages started to appear. These languages, like Lisp, Algol, Fortan, Basic, Pascal, C, C++ and Java, are processor independent. A program called a compiler is responsible for translating these general purpose languages to processor specific machine language [4][6].

These languages are labelled as higher level, because the compiler handles certain tasks automatically. These tasks include, but are not limited to:

- Processor register assignment and data exchange between memory and registers;
- Function calls and parameter passing on a call stack;
- Complex branching, conditional and looping control structures are automatically implemented with simple keywords such as *while* or *if* [6].

Source code statements no longer have a one-to-one relationship with any machine language instruction. Instead, statements form logical operations that are internally implemented by several machine instructions (which are obscured from the programmer). Thus, the programmer does not need to concern himself with architecture specific machine language, and can focus on the operation of the program instead. This implicitly improves the portability of the code between different hardware architectures. Due to this and the automatic tasks provided by the compiler, the time to develop code can be reduced significantly [6].

System languages also tend to be strongly typed. What this means is that data holders or variables are predefined by the type of data they will hold. For example: a variable defined to store integer values cannot be used to store floating-point values. An attempt to do this will most likely result in a compiler error [6].

Strongly typed languages provide the programmer with several advantages:

- Source code is easier to manage and read because the strong typing removes any ambiguity;
- The compiler can detect and warn the programmer of potential errors due to mismatched variable types;
- Because variables are of a fixed type, the compiler can optimize sections of machine code for that type.

A disadvantage of strongly typed languages, however, is that code and data locations are not interchangeable. This implies that code generation at runtime is difficult if not impossible in most cases [6].

2.4.4 SCRIPTING AND INTERPRETERS

Scripting languages can be seen as the next step in the programming evolution process. It is, however, not intended to replace system languages but rather to complement them. The purpose of scripting is to combine and chain already existing components (written in system languages) to form new applications. Hence scripting is also referred to as glue languages or system integration languages. Popular examples of scripting languages include Perl, Python, Rexx, Tcl, Visual Basic and the Unix shell [6].

An application written in a scripting language is called a script and is executed by a program called an interpreter. The interpreter reads, interprets and then executes each line of the script in sequence. Error checking also occurs when a line is interpreted. Thus every possible execution path must be executed to determine if errors are present in the script. With system languages this is done at compile time and eliminates time consuming runtime error checking [6].

In contrast to system languages, scripting languages are typeless. This leads to the following features available in scripting languages:

- Variables can change the type of data they hold at runtime;
- Code and data are interchangeable (for example, a script can write another script at runtime and then switch execution to that script);
- Variables can automatically be converted to fit its requirement (for example, a string can automatically be converted to the numeric value it represents);
- Type mismatch errors are eliminated.

These features allow variables to be used based on the value they hold and not the type representing that value. This is necessary if the system language components are to be chained together without any concern for type mismatches [6].

2.5 OPERATING SYSTEM

An operating system (OS) is a program that provides an interface layer between a user and a system's hardware and peripheral resources. It also provides the basis on top of which application software and firmware execute [7].

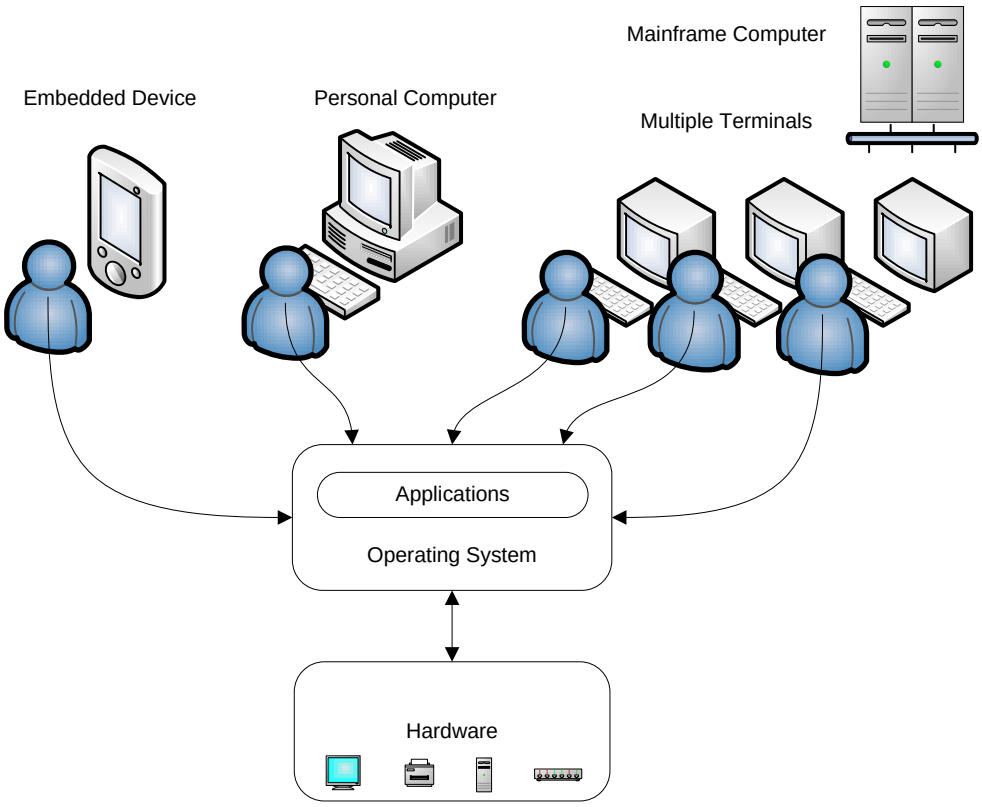


FIGURE 2-5: USER - OPERATING SYSTEM INTERFACE

The complexity of an operating system is dependent on its application area. This can range from minimal and basic support for single-user embedded devices, to very complex implementations, where a mainframe computer needs to provide equal opportunity service to multiple users via multiple terminals [7]. The application area also places different requirements on the operating system. For example, in a mainframe computer, the goal is optimal utilization of the system hardware and system efficiency. For embedded devices that rely on human interaction, the goal is rather to provide a simplistic and responsive system to the user [7].

2.5.1 OS PRINCIPLES

Even with the vast range of operating system requirements and application areas, there are certain aspects that all operating systems must address. This is true even if the choice is as simple as the decision *not* to provide a certain feature.

MULTIPROGRAMMING

In conventional systems all computer jobs are executed sequentially by a central processing unit (CPU). This approach frequently leads to idle time periods for the CPU. For example, if a job needs to wait for user input from the keyboard, the CPU will execute up to that point and then waste execution time until the user presses a key [7].

Multiprogramming is an alternative to sequential execution and provides the operating system with the ability to switch between jobs. When one job reaches a point where it needs to wait for an input or output operation (I/O), or has finished its job, the operating system can switch the CPU to execute another job. When I/O arrives, the CPU can be switched back to the waiting job and continue its execution. Therefore, CPU execution time is not wasted by waiting for a slower event to occur [7].

In operating systems that support multiprogramming an additional feature, namely time sharing or multitasking, can be provided. Multitasking is an extension of multiprogramming, where the execution of multiple applications is shared between the CPU. These switches usually occur frequently and fast enough, so that the user experiences the illusion that all the applications are executed simultaneously [7].

DUAL-MODE OPERATION

The increase in operating system requirements has led to the integration of a hardware feature that allows the CPU to operate in different modes. A minimum of two modes are usually supported, namely user mode and kernel mode [7].

The CPU is set to kernel mode whenever operating system functions are executed. These include job scheduling, hardware management and interrupt servicing. In kernel mode all CPU instructions are allowed, and the operating system is in full control of the system [7].

User applications are executed with the CPU in user mode. In this mode certain privileged instructions are prohibited from being executed by the CPU. This is to protect the operating system from misbehaving user applications, which can potentially disrupt the operating system or even the whole system [7]. Access to hardware, from user applications, is achieved through system calls provided by the operating system. These system calls execute in kernel mode and ensure that multiple applications cannot access the same hardware simultaneously [7].

SYSTEM TIMER

While the CPU is executing a user application, the operating system needs a way to reclaim control. This is achieved by setting a hardware timer to interrupt after a set period of time, before the CPU is switched to the user application. The time period can be fixed or variable, depending on the system requirements and current load. When the interrupt occurs, the operating system can decide to return execution to the user application, or to first handle required operating system jobs [7].

PROCESS MANAGEMENT

When a program is executed by the CPU, it is referred to as a process. A process can only be in one of four states [7]:

- Executing (the process is actively being executed by the CPU);
- Ready (the process is available for CPU-scheduling);
- Waiting (the process needs to wait for an event to occur before execution can continue);
- Terminated (the process has finished its execution).

In order for processes to be managed efficiently, the operating system needs to provide interfaces for [7]:

- Starting and ending both user and system processes;
- Suspending and resuming processes;
- Process synchronization;
- Process communication.

DISPATCHER

The dispatcher is the section of operating system code responsible for switching the CPU between processes. It starts by saving the current context (all register values and the CPU state of the current process), then restoring the context of the next process. The CPU is set to the correct mode of operation and execution is then directed to the point where the process was suspended (or to its first instruction for new processes) [7].

This sequence is repeated for every process switch and therefore needs to occur as fast as possible. The time needed per switch is known as the dispatch latency [7].

PREEMPTIVE OR COOPERATIVE

There are four conditions under which CPU-scheduling can take place [7]:

1. When a process switches from executing to waiting (for example, when a process needs to wait for a system call to complete, such as slow I/O access);
2. When a process switches from executing to ready (for example, when an interrupt occurs);
3. When a process switches from waiting to ready (for example, when a system call is completed);
4. When a process terminates.

The first and fourth conditions force the operating system to schedule another process (if one exists and is ready). The other two conditions provide the operating system with a choice of which process to schedule next [7].

An operating system is said to be cooperative if it only switches processes under the first and fourth conditions. Thus a process occupies the CPU until it terminates or needs to wait due to a system call. With a cooperative system, a process can only be switched at known points in its execution path [7].

However, if process switching is possible under either the second or third condition, the system is said to be preemptive [7]. Because hardware interrupts can occur at any time, process execution can also be suspended at any point. This leads to a new problem when shared data access is required between processes [7]. For example, one process needs to change the contents of a data block in memory. This process might be interrupted halfway through the change and a second process, that needs to read this same block of data, might then be activated. It will, however, read erroneous data because the first half has been updated while the second half is still old data.

MEMORY MANAGEMENT

Memory is a critical element in modern computer systems. It is the source from which the CPU reads its instructions and hardware I/O must be written into memory before it can be processed. Applications may also require additional memory for calculations or data manipulation [7].

With memory being such an important resource and with modern systems allowing multiple processes to execute concurrently, its management is critical for stable and efficient systems. Therefore the operating system is responsible for [7]:

- Keeping record of used as well as available memory;
- Allocating and releasing memory space;
- Determining which processes and or data needs to reside in memory and where.

VIRTUAL MEMORY

When no more memory is available, no more processes can be started and the system can even become unstable if running processes require more memory. A solution to this is a concept called virtual memory. It provides the system with the possibility to execute processes that are only partially in memory. If unloaded sections are needed for execution, the operating system will switch out blocks of memory to a secondary storage medium, and then load the required sections for execution [7].

This leads to the obvious advantage that programs larger than the available memory can be executed. This, however, comes at a cost, as virtual memory is difficult to implement and can incur a large decrease in performance [7].

PROTECTION AND SECURITY

Protection and security of the system becomes critical in operating systems where multiple users and/or multiple concurrent processes are allowed. Protection is based on the principle that only processes with the appropriate authority can access the requested resource (CPU, memory, files, etc.) [7].

Common examples of protection provided by an operating system are [7]:

- A process cannot infinitely maintain control over the CPU, due to the system timer;
- Hardware peripherals cannot be accessed directly by a process but must be accessed via a protected and safe operating system call;
- Processes can only access memory assigned to them. Any attempt to access memory from another process can be intercepted by the operating system and prevented.

Security can also be extended by providing users with different privilege levels of access. This means that access to system resources (executing programs, viewing or editing files, changing system date and time, etc.) can only be allowed if the current user has a sufficient privilege level [7].

REAL-TIME

An operating system is classified as real-time if the latency of certain processes can be guaranteed. The need to provide real-time support is usually only found in embedded systems. This is due to the stringent timing requirements of the application field [7]. For example, the control system of an aircraft must process its own parameters and environmental conditions within a fixed and short time period, if it is to keep the aircraft stable and on course. Additional examples include automobile anti-lock braking systems, MP3 players and digital cameras [7].

There are two main types of real-time systems, namely hard real-time and soft real-time systems. Hard real-time systems have very stringent requirements on timing and latency of certain tasks. Therefore, time-critical tasks are usually hardcoded, providing a fixed and known latency. Soft real-time systems only assigns these tasks with the highest priority. Although this approach does decrease the latency of critical tasks, this latency can still vary depending on the system load and the number of active real-time tasks [7].

2.5.2 EMBEDDED ALTERNATIVES

Every software system needs some form of operating system. To illustrate this, consider the following example: A system that alternately blinks two lights every second must be designed.

Even with this basic system, an operating system must provide functionality for at least the following:

- An output interface to switch the lights on and off;
- An interface to a peripheral timer.
- Program flow control to monitor the timer and switch lights when necessary.

Due to the responsibilities handled by the operating system, it is important to consider all available operating system alternatives when developing an embedded system. These alternatives include:

- Native code development for a specific hardware architecture;
- Low-end embedded operating systems (for example, FreeRTOS);
- High-end computer operating systems, ported to the embedded environment (for example, uCLinux).

NATIVE CODE

The first option for an embedded operating system is to manually program a basic operating system suited to the needs and processor of the specific application. This approach is usually suitable for simplistic and small systems where advanced features, such as multiprogramming, are not required. These simplistic operating systems provide at least the basic functionality with which to interface external hardware and internal peripherals.

In addition to basic I/O and interface support, program flow control must be provided by the operating system. This can be as simple as programming all tasks to execute in sequence. A powerful and popular technique is to implement a finite state machine. This implies that the execution sequence of the application is always in a known state. Every state can only be entered or exited under known conditions for that state. If this is properly implemented, the system should never be able to enter an undefined state and cause any unexpected behaviour [8].

Native operating systems suffer from certain disadvantages, as listed below:

- They are difficult to port because of the tight association with the specific hardware architecture;
- Development time and testing is time consuming, especially if advanced operating system features are required;
- Expert knowledge of the hardware architecture is required;
- In some cases, assembly language programming is the only option to implement certain features.

An advantage, however, of designing a native operating system is that it can potentially provide optimal performance. This is because it is developed for a specific hardware architecture and can make use of any specialised features provided by the processor. For this reason, implementation of a native operating system might be the only option for time-critical and real-time applications.

FREERTOS

An alternative to developing an operating system from the ground up for each application is to implement a third party operating system for the specific hardware architecture. There are many alternatives available but most of them are bound by restrictive terms of agreement or has some form of expensive licensing agreement.

FreeRTOS is a popular open source choice, with no cost implication. It is also freely available for use in commercial applications [9].

FreeRTOS is a real-time operating system that was developed specifically for embedded devices. It is easily configurable to address the requirements of most applications and provides several features [9], namely that FreeRTOS:

- Was designed to be small and with focus on ease of use;
- Can be configured for preemptive or cooperative CPU-scheduling;
- Currently supports 23 hardware architectures;
- Code is written in C and is easy to port to other architectures;
- Supports two forms of multiprogramming (namely, tasks and co-routines);
- Provides stack overflow detection;
- Includes pre-configured demo projects for most of the popular compilers;
- Supports message queues, semaphores and mutexes for inter task communication and synchronisation;
- Limits the number of tasks only by available resources;
- Supports task prioritisation;
- Provides support for microcontrollers with memory management units (MMUs).

uCLinux

Another option for hardware architectures with enough code memory and data memory is uCLinux (pronounced "you-see-linux"). This is a port[3] (specifically for embedded devices) of the popular computer operating system Linux [10].

uCLinux is a Linux distribution with some features removed to fit into the embedded application area. The most prominent feature removed is support for a hardware memory management unit. This only affects memory protection, which is manageable if it is kept in mind during software development [10].

Even with these changes, most of the Linux application base still functions perfectly. Thus, the full benefit of software developed by the Linux community is available in the embedded environment. This includes for example TCP/IP stacks, web hosting, file system management and much more [10].

[3] A *port* is the result of modifying source code, so that a software application can execute on another hardware architecture or software platform.

2.6 DEVELOPMENT TOOLS

2.6.1 COMPILER AND IDE

A compiler is a software application responsible for the conversion of programmed source code to instructions that a processor can execute. A compiler is usually used in conjunction with an integrated development environment (IDE), which provides standard tools that simplify the firmware development process. These tools include, but are not limited to:

- A text editor to edit source code;
- Syntax colour coding for easy source code editing;
- Firmware version control;
- Debugging and code-stepping;
- Error and warning indicators.

2.6.2 OPEN-SOURCE ALTERNATIVE

Development software and tools have historically been provided by third-party companies and almost always at a considerable licensing cost. With the open-source movement, several free alternatives began to appear. Over the years these open-source or free versions not only became popular, but also reliable and equal (if not surpassing) in quality with respect to their commercial counterparts.

There are many open-source alternatives available. A few popular of these are discussed further. The reader is encouraged to visit their websites for more detail [11][12][13].

ECLIPSE IDE

Eclipse is a popular IDE developed in Java. It is available for both Linux and Windows environments. Eclipse was originally designed for Java developers, but due to its flexible plug-in architecture, other languages are currently also supported, including C and C++ [11].

GNUARM

GNUARM is a set of tools (toolchain) used to compile C and C++ source code for the ARM microcontroller architecture. It functions natively on MacOS and Linux. For Windows support, a program called Cygwin must be installed on the system [12].

YAGARTO

YAGARTO ("Yet Another GNU ARM Toolchain") is a toolchain similar to GNUARM. It, however, was specifically designed to work on Windows without the need for Cygwin [13].

2.7 SUPPORT

Support is an important aspect of embedded development. This is an aspect that must be viewed from two perspectives, namely development support and maintenance support.

2.7.1 DEVELOPMENT SUPPORT

During the development process, support can help to prevent problems from being discovered too late (inherent technical risks) and can help to shorten time to market. Support is necessary (or at least desirable) from electronic component manufacturers or the developer of third party software. In this sense, the word "support" is also a collective term referring to several different forms of support, for example:

- Availability of datasheets and the quality of its contents;
- Application notes, example projects and other assistive documentation;
- Component sampling and integration support;
- Manufacturer support for production-based decisions;
- Support when new software drivers or modules are developed;
- Change control when changes are made to software toolchains, etc.

2.7.2 MAINTENANCE SUPPORT

Once development reaches the production phase, maintenance becomes an integral part of the project. This not only includes logistics and support to ensure a continued production process, but also a responsibility of providing support to any clients or users of the product. These maintenance responsibilities include, but are not limited to [1]:

- Initial installation or setup of the product;
- Training and certification of any operators;
- Support documentation;
- Software upgrades and maintenance;
- Change control and associated support;
- Spare parts or repair services;
- Disposal of retired units.

2.8 SUMMARY AND CONCLUSION

This chapter provided an overview of the philosophies of PLC controllers, operating systems, interpreters and development support. These philosophies are important for the development of a real-time interpreter. The following important design requirements are thus confirmed from the literature study:

- The interpreter must be real-time in the sense of minimising any latency during execution;
- Inputs and outputs must be synchronised so that changes do not take place during execution (must be deterministic in nature);
- Program sequences must execute sequentially, as with existing PLC programs;
- The interpreter should run on an OS that is freely available, reliable, and portable;
- Development tools should be supported and should be low cost.

The recent improvements in processing speed and decrease in cost have made it possible to implement a real-time interpreter on an OS that supports real-time applications. This should be implemented on a cost-effective microcontroller or microprocessor that will allow for fast execution with good support and availability.

CHAPTER 3
PRELIMINARY DESIGN OF AN INTERPRETER

3.1 INTRODUCTION

In order to design an interpreter that addresses the requirements of a PLC system, it is necessary to first consider all the steps in a typical PLC development cycle. This is achieved by performing a functional analysis of a typical PLC project life cycle. After this analysis, a functional architecture to realize these functions, with an interpreted system, is presented and explained.

3.2 FUNCTIONAL ANALYSIS OF A TYPICAL PLC PROJECT CYCLE

In the average PLC project life cycle, the detailed design stage requires the bulk of the work and development time. A typical life cycle of a PLC project is shown in Figure 3-1, where the process is shown from the definition of need up to the project close-out.

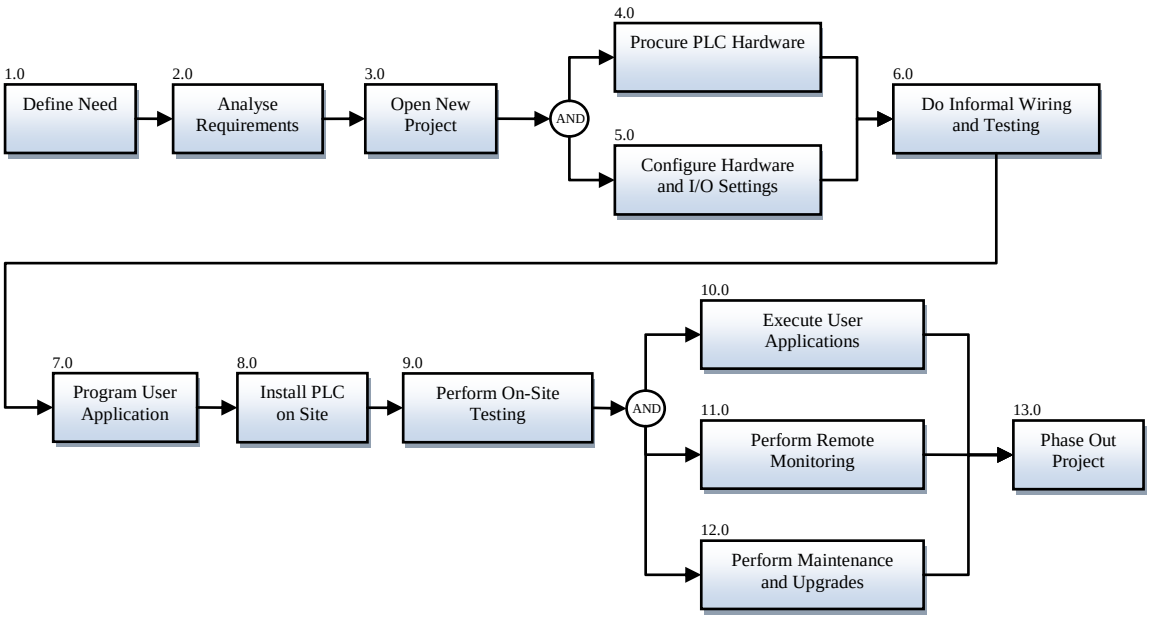


FIGURE 3-1: FUNCTIONAL ANALYSIS

The functions considered in this work will be described in more detail in the sub-sections that follow. An operational function is denoted by O/F, while a maintenance function is denoted by M/F.

3.2.1 OPEN NEW PROJECT (O/F 3.0)

This function is initiated by the user when a new project is started. It will be handled by some form of computer software, providing the user with a blank workspace and interface to configure and program the PLC. This is done on the computer by a programmer who has to program the PLC. This part of the system has been fully implemented but is not relevant to an understanding of this work and will therefore not be explained in detail. It is, however, shown for the sake of completeness.

3.2.2 PROCURE PLC HARDWARE (O/F 4.0)

This process is dependent on the user application as well as the company's specific process for acquiring hardware. The procurement of hardware does not form part of the focus of this project and will therefore not be discussed in detail.

3.2.3 CONFIGURE HARDWARE AND I/O SETTINGS (O/F 5.0)

PLC configuration is a two-fold process. Firstly, it requires the user to decide which inputs and outputs are connected to specific peripheral devices. This task is application specific and serves to show that inputs and outputs should be configured in "real-world terms" so that users can understand the context of the PLC configuration.

The second aspect consists of the user configuration of the software project to reflect the actual hardware connections. This task includes the sensible naming of I/O ports for clarity and ease of reference (this correlates with the definitions of the devices that are connected to the PLC). Translation of the operational names of I/O ports is done by the high-level PC software to provide low-level hardware specific references for later use during PLC program execution. It suffices to know that this translation must take place and that it will be done by high-level PC software.

3.2.4 DO INFORMAL WIRING AND TESTING (O/F 6.0)

After I/O ports have been assigned, the ports can be connected to devices in a test environment. This is not the final wiring of the application and is only intended for program testing and system verification.

3.2.5 PROGRAM USER APPLICATION (O/F 7.0)

With a test setup wired and configured in software, the programming of the PLC can start. The following functional flow provides a more detailed breakdown of this function (see Figure 3-2).

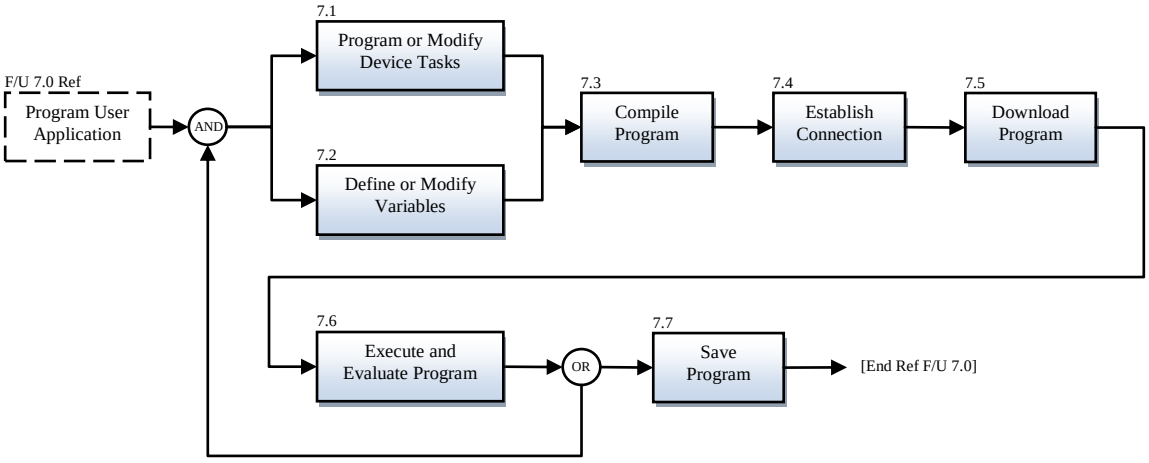


FIGURE 3-2: FUNCTIONAL ANALYSIS: F/U 7.0

The first step is to program the actual tasks that must be performed by the PLC. This usually requires the definition of variables that are used in these program tasks. It is important to know that the PLC operational functions are defined by rules, as derived from the operational analysis performed by the user for the system. These rules must be constructed in the form of events that perform an appropriate action when certain conditions are met.

These user programmed events can then be compiled to an executable format, which can be interpreted by the PLC. After a communication link has been established, this executable program can be downloaded to the PLC.

The PLC executes the program, while the user verifies that all required PLC control actions are taken. This cycle can be repeated until the user is satisfied with the program. The final step is to save this program to the PLC so that it will execute automatically after the PLC has been powered up.

3.2.6 INSTALL PLC ON SITE (O/F 8.0)

This step is application specific and will not be discussed in detail. However, it does consist of the installation and final wiring of the PLC system and shows that the user needed to perform a proper operational analysis before the PLC was programmed.

3.2.7 PERFORM ON-SITE TESTING (O/F 9.0)

This is the final testing stage to verify that the PLC performs all desired actions correctly. It should be evident that a good initial configuration will reduce the complexity of debugging at this stage since the program reflects the operational configuration.

3.2.8 EXECUTE USER APPLICATIONS (O/F 10.0)

This function is responsible for executing the currently saved user program. This entails the evaluation of all user programmed events (condition-action pairs).

3.2.9 PERFORM REMOTE MONITORING (O/F 11.0)

This is both a maintenance and operational function in which the user can remotely connect to the PLC and monitor its operation. This may include the current state of configured I/O ports and program variables. The fact that the PLC in this work has a GSM module (or other TCP/IP connection) means that remote monitoring is fairly easy.

3.2.10 PERFORM MAINTENANCE AND UPGRADES (M/F 12.0)

When requirements change (or an error is discovered), the PLC program might need to be modified and upgraded. This can also include hardware changes or additions.

3.3 FUNCTIONAL ARCHITECTURE

An overview of the system functional architecture is shown in Figure 3-3

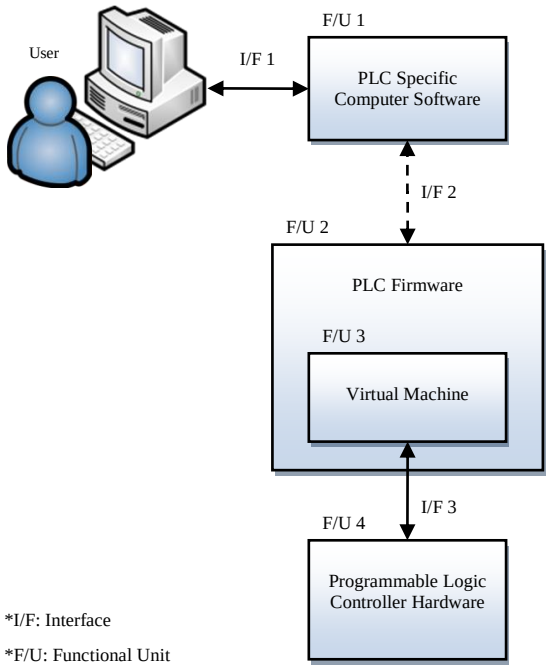


FIGURE 3-3: FUNCTIONAL ARCHITECTURE

The user can configure and program the PLC using specialised computer software (F/U 1). The PLC specific computer software must be capable of establishing a virtual connection (I/F 2) with the PLC. This is required in order to exchange data for programming and remote motoring. This virtual connection must be independent of the physical medium used (for example: USB, serial port or GSM network) and will thus form a transparent link between the PLC computer software and PLC.

An operating system will form the base of the PLC firmware (F/U 2) and provide a reliable layer between the PLC hardware (F/U 4) and the virtual machine (F/U 3). The virtual machine is the core of the system and is responsible for executing all user programmed tasks.

3.3.1 PLC SPECIFIC COMPUTER SOFTWARE (F/U 1)

Figure 3-4 shows a second level overview of the functional architecture for the PLC specific computer software unit (F/U 1).

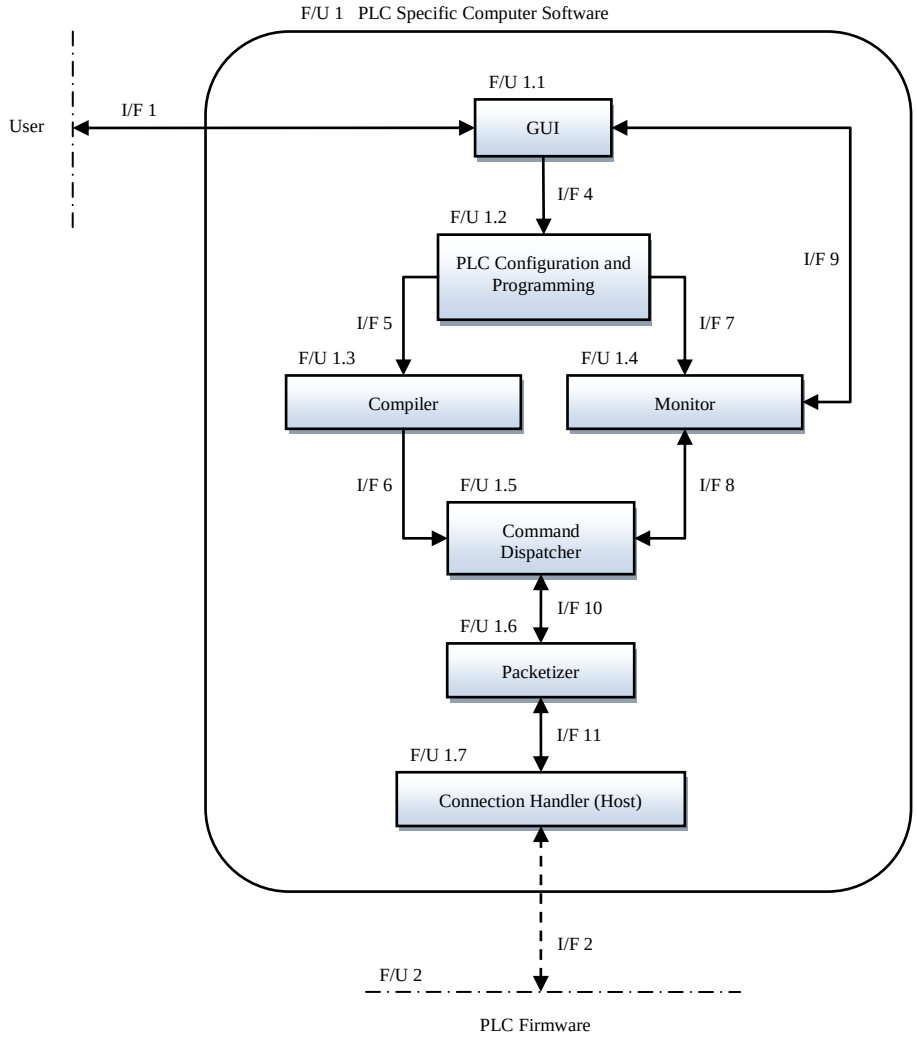


FIGURE 3-4: FUNCTIONAL ARCHITECTURE OF F/U 1

The primary purpose of the PLC specific computer software is to provide a user-friendly platform for configuring, programming and monitoring the PLC. This is achieved by providing a graphical user interface (GUI) (F/U 1.1), through which the user has access to configuration options (F/U 1.2), a programming interface (F/U 1.2) and information displays for monitoring data.

The compiler (F/U 1.3) converts the user-programmed PLC applications from an easy human understandable language to machine instructions that can be executed by the PLC's virtual machine. The monitor (F/U 1.4) is a user-configurable unit that can request the state of PLC variables or I/O and display their results in the GUI.

The command dispatcher (F/U 1.6) is a unit that communicates with, and controls, the PLC via a predefined set of commands. These commands (I/F 10) include, for example, requests from the monitor for data and I/O states. Commands are also used to download user programmed applications to the PLC.

The packetizer (F/U 1.6 and F/U 2.2) converts the raw commands to and from a predefined data packet format. Each packet contains information about the size and destination of the data as well as error checking information.

The connection handler (F/U 1.7) is responsible for establishing and maintaining a transparent link (I/F 2) between the computer software and the PLC. The user must select the desired connection medium to use from a list of supported types (for example: USB, serial port, GSM network). Irrespective of the choice, the connection handlers (both F/U 1.7 and F/U 2.1) must mask the medium as a generic virtual connection. This helps to objectify the firmware and improves modularity, thus providing a simple interface and opportunity to add support for other connection mediums in the future.

3.3.2 PLC FIRMWARE (F/U 2)

Figure 3-5 shows a second level overview of the functional architecture for the PLC firmware unit (F/U 2).

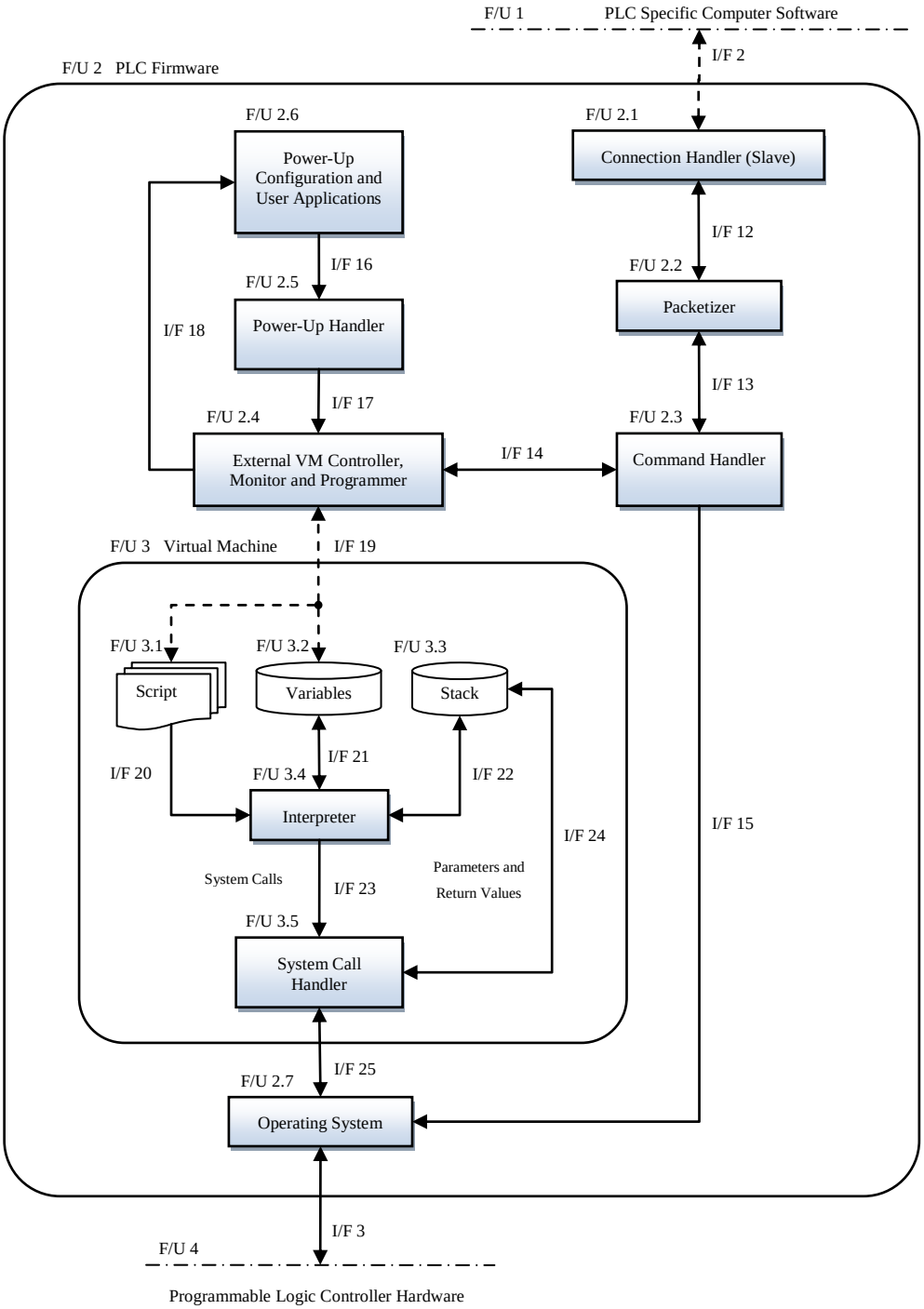


FIGURE 3-5: FUNCTIONAL ARCHITECTURE OF F/U 2

The connection handler (F/U 2.1) and packetizer (F/U 2.2) are similar to their counterparts on the computer side, as described in the previous section. The command handler (F/U 2.3) is responsible for decoding and executing the commands sent by the command dispatcher (F/U 1.5). Depending on the command, it can access either the underlying core functionality or manipulate the virtual machine via the VM controller (F/U 2.4).

The VM controller has direct access to all aspects of the virtual machine. These include the following:

- Starting, suspending and stopping the virtual machine;
- Programming user applications (scripts);
- Reading user applications back to the PLC computer software;
- Modifying variable values;
- Reading variable values for monitoring;
- Saving PLC configurations and user applications

The power-up handler (F/U 2.5) takes care of initialization when the PLC is powered up. It programs and starts the virtual machine from the saved PLC configuration and user applications (F/U 2.6).

The operating system (F/U 2.7) is a critical component of the PLC firmware. It is responsible for providing low-level firmware functions on which the virtual machine will run to ensure a stable working platform. The OS must also provide a safe and managed interface to the PLC hardware and system resources. This is achieved in conjunction with the hardware abstraction layer (HAL) interface (I/F 3), which provides low level firmware functionality to access the hardware directly.

3.3.3 VIRTUAL MACHINE (F/U 3)

Figure 3-6 shows a second level overview of the functional architecture for the virtual machine (F/U 3).

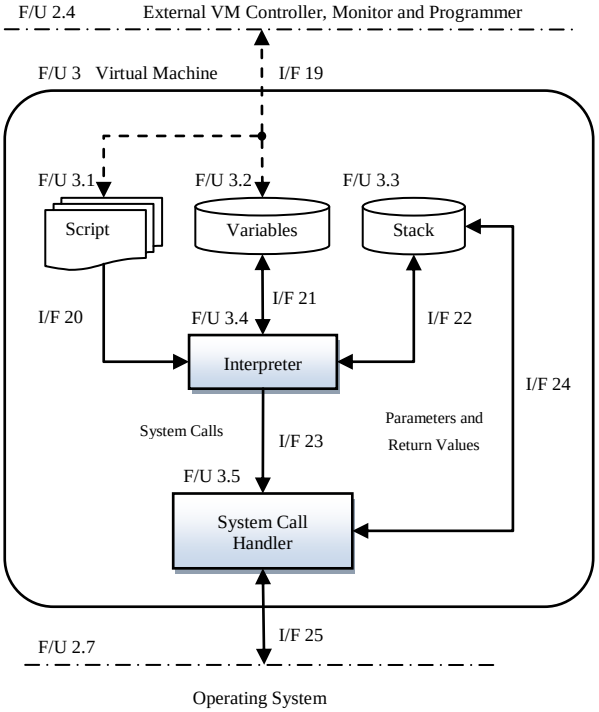


FIGURE 3-6: FUNCTIONAL ARCHITECTURE OF F/U 3

The purpose of the virtual machine (VM) is to provide a scripting engine that is capable of interfacing with all underlying hardware functionality. The VM must also be programmed in a language independent of the specific hardware architecture. This can be achieved due to the hardware abstraction layer (I/F 25) that effectively guarantees portability (of course, changing the hardware platform requires a new hardware abstraction layer for the OS and a new System Call Hander in the VM, as shown above).

The VM emulates the same components as found in actual hardware processors, since the processor's internal functionality is based on tried and tested principles and philosophies – this gives robustness. These VM elements include:

- A stack (F/U 3.3) used to temporarily store data in a LIFO[4] buffer;
- Memory space for variables (F/U 3.2) and data holders;
- Machine instruction set that can be chained to form programs (scripts);
- A processor (F/U 3.4) to execute these scripts (F/U 3.1)

An additional feature, required by a virtual machine, is a mechanism to interface the actual hardware on which it is running as if it were its own. This ability is provided by the system call handler (F/U 3.5) and is also the only VM unit that needs to be modified to support different hardware architectures.

3.3.4 PLC HARDWARE (F/U 4)

Figure 3-7 shows a second level overview of the functional architecture for the PLC hardware unit (F/U 4).

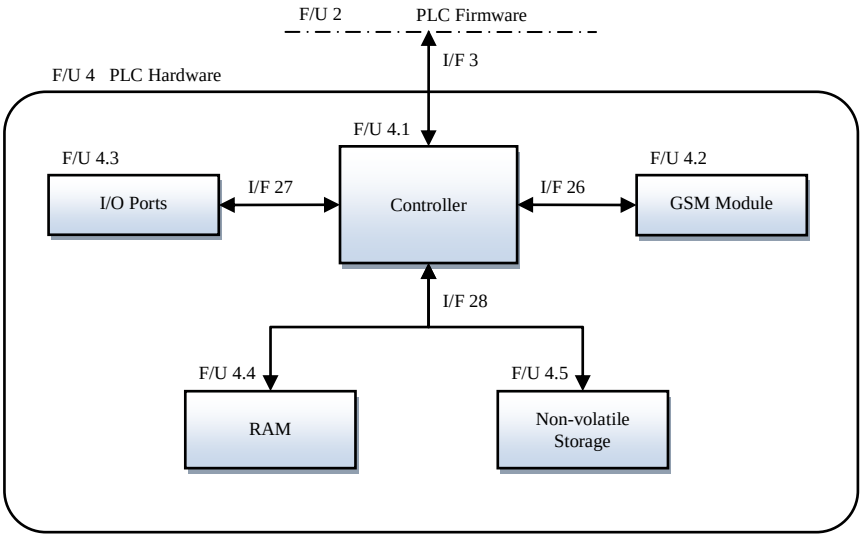


FIGURE 3-7: FUNCTIONAL ARCHITECTURE OF F/U 4

[4] LIFO stands for Last-In-First-Out and signifies that data is inserted sequentially and removed in the inverse order they were added.

The controller (F/U 4.1) is the central unit of the PLC hardware. It provides all required processing power and also controls external peripherals. The GSM module (F/U 4.2) provides GSM capabilities to the PLC. The non-volatile storage (F/U 4.5) stores the PLC firmware as well as all virtual machine configurations, user applications, and operational configuration data – this is important since this functionality is not available on conventional PLCs. The RAM (F/U 4.4) stores, as usual, all temporary data, runtime variables, etc. The I/O ports (F/U 4.3) simply serve as digital or analog input and output ports.

3.4 INTERFACE SUMMARY

Table 3 provides a summary and short description of all functional interfaces.

TABLE 3: INTERFACE SUMMARY

I/F no.	I/F Type			Description
	Electronic	Software	User	
1			x	Interface between the user and the computer software
2	x			Communication channel between the computer and the PLC
3		x		Hardware abstraction layer (HAL) between the PLC hardware and firmware
4			x	User changes to PLC configuration and programming
5		x		High-level user-programmed code sent to the compiler
6		x		Low-level interpreter code (script) sent to the PLC via the command dispatcher
7		x		PLC configuration data sent to the monitor
8		x		Requests sent to the PLC, via the command dispatcher, for monitor data and the results returned to the monitor
9			x	Configuration from user of what to monitor and the monitored results to the GUI
10		x		Raw commands and data sent to, or command responses received from, the packetizer
11		x		Packeted data sent to or received from the PLC via the connection handler
12		x		Packeted data received from or sent to the computer via the connection handler
13		x		Received commands and data to the command handler or command responses to the packetizer
14		x		Control signals or programming data to the virtual machine or monitored results from the virtual machine.
15		x		Control signals to interface the PLC core directly
16		x		Saved configurations and user applications sent to the power-up handler
17		x		Programming data and initial state configurations to the virtual machine
18		x		Current virtual machine configuration and programmed applications to be saved
19		x		New or modified applications and variables to the virtual machine or monitored results from the virtual machine
20		x		Scripted code to the interpreter for execution
21		x		Read variable data or data to be written in variables
22		x		Data to and from the stack
23		x		System calls from virtual machine to its handler
24		x		Parameters used in system calls or returned values from system calls
25		x		Requests from system call handler to underlying architecture and operating system
26	x			Communication channel between controller and GSM module
27	x			Signals between I/O and controller
28	x			Data bus between controller and memories

3.5 RESOURCE ALLOCATION

The following table depicts the resource allocation of the system, which maps the functions from the functional flow to the functional units in the system architecture. The functional units from the system architecture are assigned to rows, while functions from the functional flow are assigned to columns.

TABLE 4: RESOURCE ALLOCATION

		O/F 3	O/F 5	O/F 6	O/F 7.1	O/F 7.2	O/F 7.3	O/F 7.4	O/F 7.5	O/F 7.6	O/F 7.7	O/F 8	O/F 9	O/F 10	O/F 11	M/F 12
F/U 1	F/U 1.1	x	x		x	x									x	x
	F/U 1.2		x		x	x	x		x						x	x
	F/U 1.3						x									x
	F/U 1.4														x	
	F/U 1.5							x	x		x				x	x
	F/U 1.6							x	x		x				x	x
	F/U 1.7							x	x		x				x	x
F/U 2	F/U 2.1							x	x		x				x	x
	F/U 2.2							x	x		x				x	x
	F/U 2.3							x	x		x				x	x
	F/U 2.4								x		x		x	x	x	x
	F/U 2.5												x	x		
	F/U 2.6										x		x	x		x
	F/U 2.7									x			x	x		
F/U 3	F/U 3.1								x	x	x		x	x		
	F/U 3.2									x			x	x	x	
	F/U 3.3									x			x	x		
	F/U 3.4									x			x	x		
	F/U 3.5									x			x	x		
F/U 4	F/U 4.1									x			x	x		
	F/U 4.2									x			x	x		
	F/U 4.3			x						x		x	x	x		
	F/U 4.4									x			x	x		
	F/U 4.5												x	x		

3.6 SUMMARY

This chapter provided the preliminary design of an interpreter system for a PLC. A functional analysis was performed on a typical PLC project which led to the functional architecture for this project. These sections include an overview of the system, consisting of the PLC computer software *(including high-level programming language, compiler, and monitor)* and the PLC firmware *(including virtual machine, scripts, and operating system)*. A high-level overview of the hardware was also given.

As a final contribution, the interfaces between functional units are summarized and described in table form, followed by the resource allocation table providing the link between function and architecture.

CHAPTER 4
DETAIL DESIGN

4.1 INTRODUCTION

A detail design usually includes hardware and software design. This is done down to component and detailed code level. The detail design of the PLC hardware was done but is not pertinent to an understanding of this work. As a result, it will not be discussed in detail in this document.

This chapter gives the detailed design of the project elements, which include the design and programming of the PLC computer software *(including high-level programming language and compiler but excluding GUI)* and the design and programming of the PLC firmware *(including virtual machine, VM instruction set, interpreter, and scripts).*

4.2 PLC HARDWARE (F/U 4)

The PLC, for the sake of this research and development, consists of a microcontroller for processing and control, a GSM module, analog inputs and digital inputs and outputs. *(Note that only the interpreter interface with inputs and outputs is discussed in this document. Additional PLC functionality, such as GSM services, pulse counters, timers, etc., has been implemented but their interaction with the interpreter is identical to that of I/O. Therefore, a detailed explanation would be redundant and has been purposely left out of this document).*

Different processors exist on the market, but few have processors powerful enough to effectively run Linux or very large multi-media applications. *The client specified the use of a Telit GSM module due to the powerful ARM processor on the module.*

Telit, a company that designs and manufactures GSM modules, has developed the GE863-PRO[3] GSM module that contains a fully configured ARM9 microcontroller. This system has the following specifications [14]:

- AT91SAM9260 microcontroller from Atmel;
- Configured at 100 MIPS[5];
- 4 MB serial flash;
- 8 MB SDRAM.

An additional motivation to use this module is that Telit provides firmware for two different modes of operation. The first is a preconfigured project to program the microcontroller natively. They also include libraries to access the embedded GSM part of the module.

The second mode is similar to the first, but it is built on a fully functioning embedded Linux distribution. This eliminates the need to develop a real-time operating system for this project. However, it was still deemed necessary to test the capability of Linux to perform real-time tasks – this test was done and is discussed in Chapter 5.

4.3 PLC SPECIFIC COMPUTER SOFTWARE (F/U 1)

Programming of the computer software was done in C# and Microsoft .NET. The development environment used was Microsoft Visual Studio 2008 Express Edition, which is free for student use [15]. Note that the detail design of the GUI was completed but is also not pertinent to an understanding of this work (the interpreter). Nonetheless, it is briefly discussed for the sake of completeness.

[5] MIPS or Million Instructions Per Second, specifies the number of instructions a processor can execute in one second. This is an indication of processing power.

4.3.1 GRAPHICAL USER INTERFACE

For an understanding of the GUI's relationship to the rest of this work, the GUI should only be thought of as the interface through which the user controls, and interfaces with, the PLC.

The GUI provides the user with the following features:

- Naming and configuration of I/O ports;
- Definition of variables;
- Programming of application events;
- Monitoring of data inside the PLC (also from peripheral IO devices);
- A clear indication of the PC ⇔ PLC connection status.

4.3.2 PLC CONFIGURATION AND PROGRAMMING

This is the primary interface provided by the GUI. It can be divided into two sections, namely configuration and programming.

The configuration defines all the information that is used by the PLC. This includes the size of an inserted SD-card and the cellular number of the SIM card used in the GSM module. For this project, however, only the I/O ports are discussed in detail. This is because an understanding of their relationship with the interpreter can be generalized for all other interfaces.

A new programming language was defined to address the requirements for multi-media and code flexibility. This approach also aims to simplify the programming process by converting "classical" programming techniques into a language that is intuitively understandable by humans.

All user applications are structured into a fixed format of several events. These events are executed in sequence. Each event consists of an if-then-else clause. The "then" is a list of actions to be taken if the "if" conditions evaluate to true. These conditions can be chained, to form complex expressions, using logic "and" and "or" operators. An optional "else" list of actions can be defined to be taken when the conditions are not met. A representation of such a user application is shown in Figure 4-1.

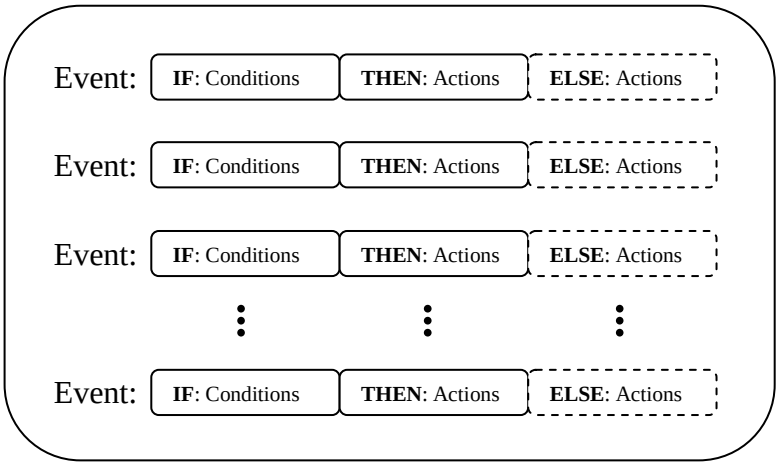


FIGURE 4-1: EXAMPLE USER APPLICATION

The available conditions and actions are predefined and associated with a specific interface, where an interface is any part of the PLC (logical or physical) that can form a condition or perform an action. For example, a digital input is an interface that can be used to form a condition (if digital input is high), but it cannot perform an action. A digital output can form a condition (if digital output is low) as well as perform an action (digital output is set to high).

An understanding of the basic I/O interfaces can easily be extended to other interfaces, such as an LCD, SMS handler, timer or counter. The interfaces defined for this discussion are digital inputs and outputs, analog inputs and variables.

A detailed list of these interfaces is provided in Appendix A. This is a comprehensive list and includes the available conditions and actions for each interface.

An additional feature available to the user is the option to specify the times at which an event must be evaluated (for example, every three seconds, or only on weekdays).

4.3.3 COMPILER

The purpose of the compiler is to convert the user-programmed events, into a PLC executable script.

A powerful technique implemented in PLC ladder logic programs, is that all inputs are read at the start of the program and all outputs are written out only at the end of the program [5]. It has been decided to follow this approach in order to guarantee I/O-synchronization, and hence a stable and deterministic program flow.

Therefore, the compiler will encapsulate the user-programmed events between the "Buffer Inputs" and "Write Outputs" system calls[6]. Any other system calls to inputs will reference these buffered inputs instead of the state of the physical input port. Similarly, any system calls to change the state of an output will only take effect after the "Write Outputs" system call.

Because the events are executed sequentially, their compilation can be considered separately. The basic principle behind each event is to first evaluate all conditions and then jump to the code section that performs the appropriate actions ("then" or "else"). This is demonstrated in Figure 4-2.

To allow events to execute only at certain times or periodic intervals, an additional condition and jump is inserted before those events. The condition is first evaluated, and if this condition fails the jump skips the associated event and execution continues at the next instruction.

[6] System calls are explained later in this chapter, but it can be thought of as the mechanism, with which the virtual machine can access the underlying system functionality (for example, direct access to hardware).

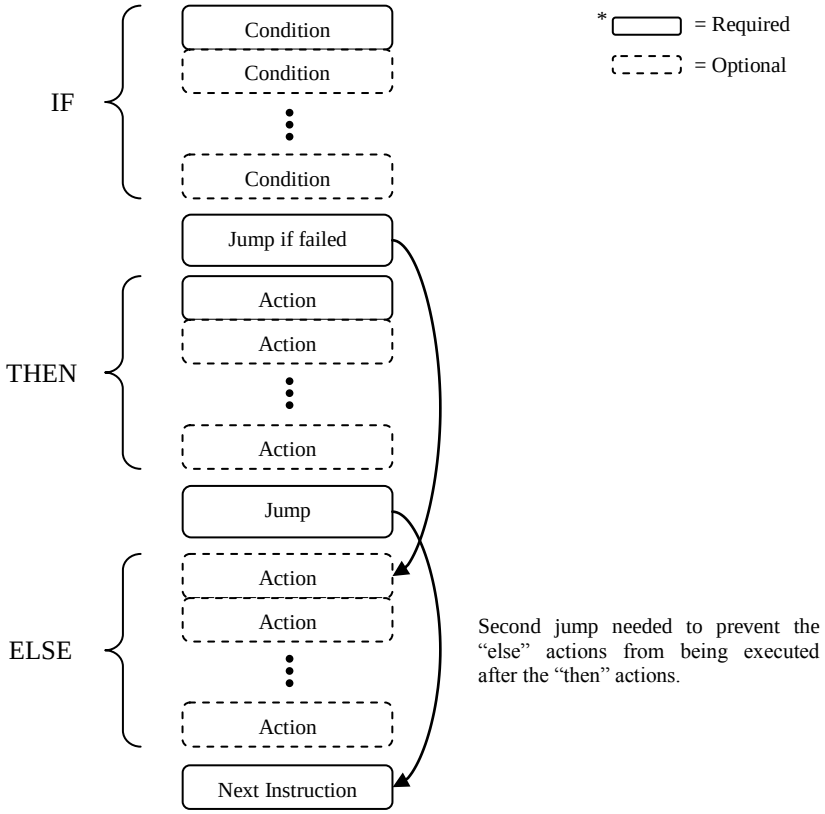


FIGURE 4-2: EVENT COMPILATION APPROACH

Another feature of the script is that it can be used to configure the virtual machine itself. This is achieved through special system call functions as explained later on. The compiler implements this feature by placing all configuration instructions at the start of the script. The actual user program is then appended, followed by a jump instruction to the start of the user program. This effectively causes the configuration to be executed once on start-up, followed by the user program for the rest of the VM operational time. This is shown in Figure 4-3.

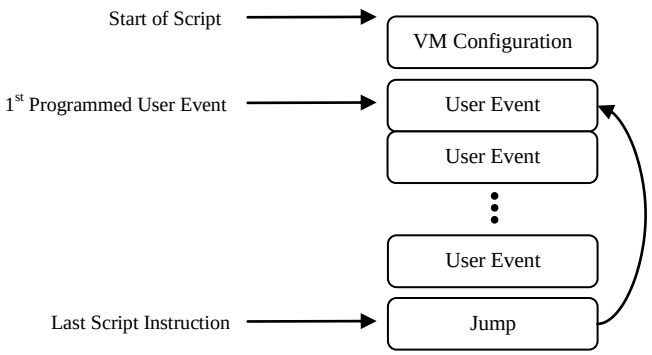


FIGURE 4-3: VM CONFIGURATION IN SCRIPT

4.3.4 MONITOR

The monitor is responsible for querying the PLC's internal properties and displaying the results on the GUI. This is achieved by periodically sending a request to the PLC for the desired data. This can include requests for:

- I/O states;
- Values of VM variables;
- VM performance criteria (for example: program cycles per second, total time active, etc.);

The actual format of these requests is described in the "Command Dispatcher" section.

4.3.5 PACKETIZER

The packetizer is responsible for packaging data in a reliable format for transmission. The format of a packet is shown in Figure 4-4.

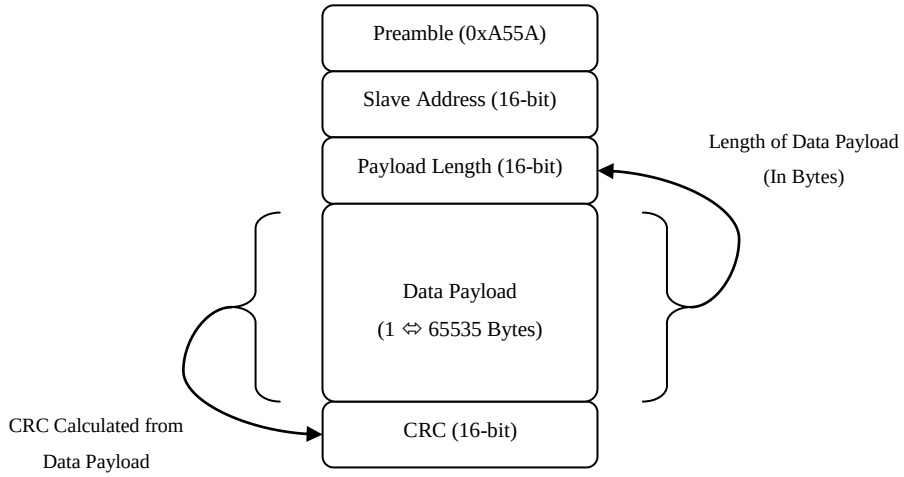


FIGURE 4-4: SERIAL PACKET FORMAT

A preamble is placed at the start of the header, so that the receiver can recognise the start of a packet. The chosen sequence is 0xA55A (hexadecimal), which results in a balanced number of binary ones and zeroes (1010 0101 0101 1010).

A slave address is inserted for future support of chained PLCs. For example, three PLCs can be connected in series, with the first directly connected to the computer. A packet sent, with a slave address of one, will arrive at the first PLC, address of two at the second PLC and so forth. Thus the computer software will be capable of establishing a direct virtual link to all three PLCs. With the slave address zero, the packet is broadcasted to all connected devices.

The next field specifies the number of data bytes that follow. This allows a packet size with a data payload from 1 to 65535 bytes.

The packet is ended with a 16-bit CRC number, calculated from the data payload. This CRC format is fully described by the following parameters:

- Width of 16 bits;
- Polynomial 0x8005 (hexadecimal);
- Reflected input (least significant bit first);
- Initial CRC value of 0.

To verify that the CRC is calculated correctly, a test sequence of ASCII characters "123456789" (0x31 0x32 0x33... hexadecimal) must end with a CRC of 0xBB3D (hexadecimal).

4.3.6 COMMAND DISPATCHER

The command dispatcher is the unit responsible for sending commands and requests to the PLC. These commands will be sent in the data payload section of the packets created by the packetizer. Multiple commands may occur within a single packet, in order to minimize the data transferred, due to the packet formatting overhead.

There are three different command formats, each predefined by the command type. The reason for implementing these three formats is to provide a flexible command-based platform, without the need for extra formatting fields. Because each command has a predefined format, both the PLC and computer software knows how to interpret the data, without extra negotiations. Thus, effectively eliminating unnecessary data transfer.

The following figure shows these three command formats, followed by a description and example of each.

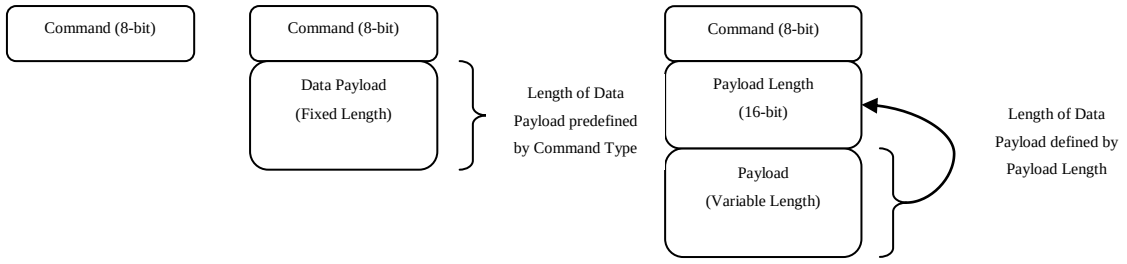


FIGURE 4-5: COMMAND AND COMMAND-WITH-PAYLOAD FORMATS

The first command format consists of just a command code (8-bit value indexing the command). These types of commands do not require any additional data to fulfill their task and are usually control commands, such as a hardware reset request.

The second format has a fixed length data payload appended after the command code. The actual length of the data payload is predetermined by the type of command. This is possible for commands that always send the same number of data bytes, such as commands to configure the time.

The last format requires a variable length data payload, such as when the PLC needs to be programmed. Therefore, these command types have an extra 16-bit length field, specifying the number of payload bytes that are to follow.

Note that no error checking is included in the command structure, since a successful reception of the packet (which included error checking) implies that no error has occurred within the data.

For a complete list of implemented commands, please refer to Appendix B.

4.3.7 CONNECTION HANDLER (HOST)

The connection handler on the host PC is primarily responsible for establishing a virtual link between the computer software and the PLC. The link is referred to as "virtual", because the actual connection medium used should be masked once a physical connection has been established. Thus, irrespective of using USB, serial port or GSM network, it should only be known that a connection exists and that data can be transferred.

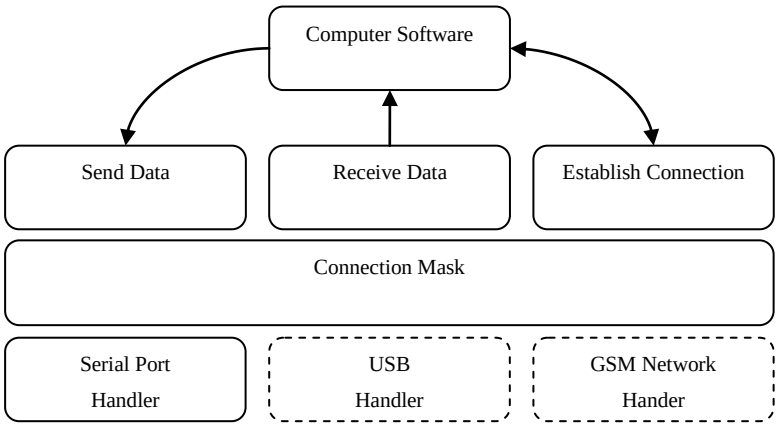


FIGURE 4-6: CONNECTION MASK

The connection mask is a software unit that provides certain interfaces to the rest of the computer software (as shown in Figure 4-6). Of these interfaces, only "Establish Connection" makes mention of a specific medium. This is because the user needs to specify the desired connection medium to use. Once a connection is established, the other interfaces become available. These are generic and not bound by any medium. For example, "send data" will simply send the desired data across the currently active connection. Similarly, any received data (from any connected medium) will be made available to the computer software.

Behind the connection mask are the software units that manage a specific connection medium. The connection with the PLC is established by sending the "Test Connection" command to the PLC and waiting up to 10 milliseconds for a response. This is repeated 100 times before assuming failure, or until a successful response is received. The actual transmission and reception of serial data is managed by the .NET library [15].

4.4 PLC FIRMWARE (F/U 2)

The firmware development was done with the development environment provided by Telit. It consists of a Telit customized Eclipse for the IDE and a Linux compiler toolchain for developing firmware for their Linux distribution. This compiler toolchain is intended for the C programming language. Although this development environment can manually be installed and configured, Telit greatly simplified this process with an automatic installer [16].

4.4.1 CONNECTION HANDLER (SLAVE)

Similar to the connection handler of the computer software, this module is also responsible for providing a virtual link to the computer software. The serial port is the only connection medium implemented, but due to a masking layer, support for other mediums can easily be added in the future. Contrary to the host side connection handler, the slave always keeps the connection open, listening for any commands. Any received data is sent to the command handler via the packetizer.

4.4.2 PACKETIZER

The PLC side packetizer is identical in functionality to the packetizer on the PC software side. It only differs in the hardware platform on which it operates, namely an embedded environment for the PLC.

4.4.3 COMMAND HANDLER

Commands that were successfully decoded by the packetizer are sent to the command handler. This unit performs the requested action and sends an acknowledgment back to the computer software. These actions are mostly to program, manipulate or observe the virtual machine via the VM controller.

This unit also has direct access to the operating system and underlying core functionality. These features (for example, directly manipulating output ports, reading input port states, restarting the processor, etc.) are accessed through the same command structure as mentioned above. However, these commands are used through the development process and for debugging purposes. The user will not have direct access to these functions.

For a complete list of supported commands, refer to Appendix B.

4.4.4 POWER-UP CONFIGURATION AND USER APPLICATIONS

This unit is a non-volatile storage medium that holds the user programmed application and configuration of the PLC.

An SD-card and on-board flash memory were provided on the PLC hardware. The SD-card holds user data and the flash memory holds user programs.

The information is saved at a fixed offset near the end of the flash memory. It starts with a field specifying the number of bytes to follow and ends with a 16-bit CRC code. The CRC algorithm is the same as used in the packetizer. If the CRC does not match, the PLC will not attempt to use this data and will remain idle. Figure 4-7 shows the format and location of the user application.

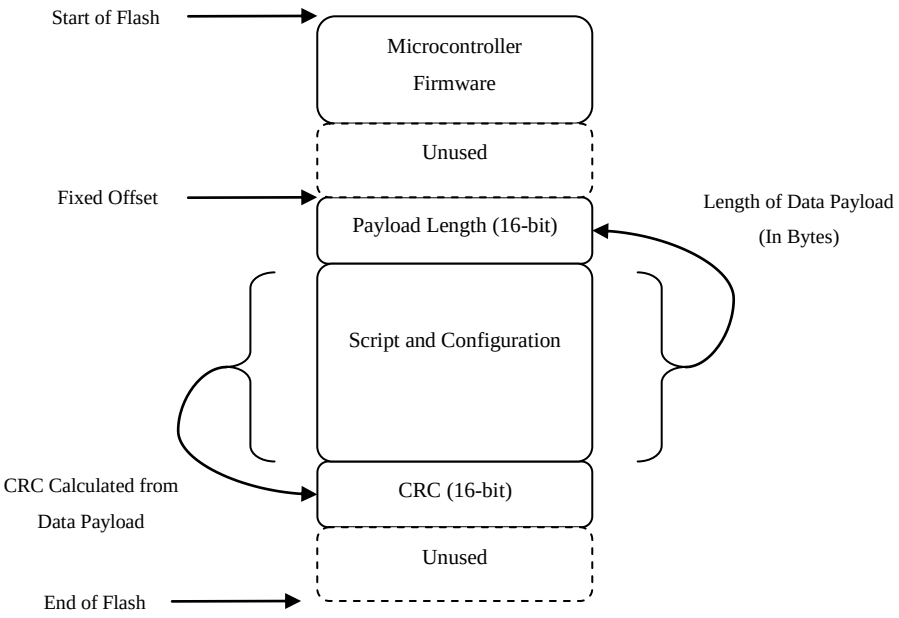


FIGURE 4-7: SCRIPT LOCATION IN FLASH

4.4.5 POWER-UP HANDLER

The power-up handler is a small piece of firmware that is executed only once every time the PLC is switched on. It is responsible for reading any saved configuration and user applications from a non-volatile storage medium and to setup the virtual machine to this saved state. Afterwards the PLC operates normally.

The actual configuration of the virtual machine is handled by the virtual machine controller.

4.4.6 EXTERNAL VM CONTROLLER, MONITOR AND PROGRAMMER

This software unit provides functionality to directly manipulate or observe the virtual machine. This is achieved by providing software interfaces for the following tasks:

- Starting, suspending and stopping the virtual machine;
- Programming user applications (scripts);
- Reading user applications back to the PLC computer software;
- Modifying variable values;
- Reading variable values for monitoring;
- Saving PLC configurations and user applications

4.4.7 OPERATING SYSTEM

As was mentioned earlier, Telit has already implemented a Linux distribution on the GE863-PRO^3 module [16]. Thus, no further development is needed for the operating system and all firmware modules were developed on this platform.

It is important to note that a FreeRTOS implementation was also evaluated and that these details are provided in Chapter 5.

Also note that most of the firmware modules will function on both operating systems with no or minor changes. This is due to an object-orientated approach that was followed during firmware development.

4.5 VIRTUAL MACHINE (F/U 3)

The design in this section was influenced by techniques and ideas discussed in the following sources [17][18][19].

The purpose of the virtual machine is to provide a scripting engine that is capable of interfacing with all underlying hardware functionality. The VM must also be programmed in a language independent of the specific hardware architecture. In order to achieve this, several functional units are required. These units are described in the rest of this section.

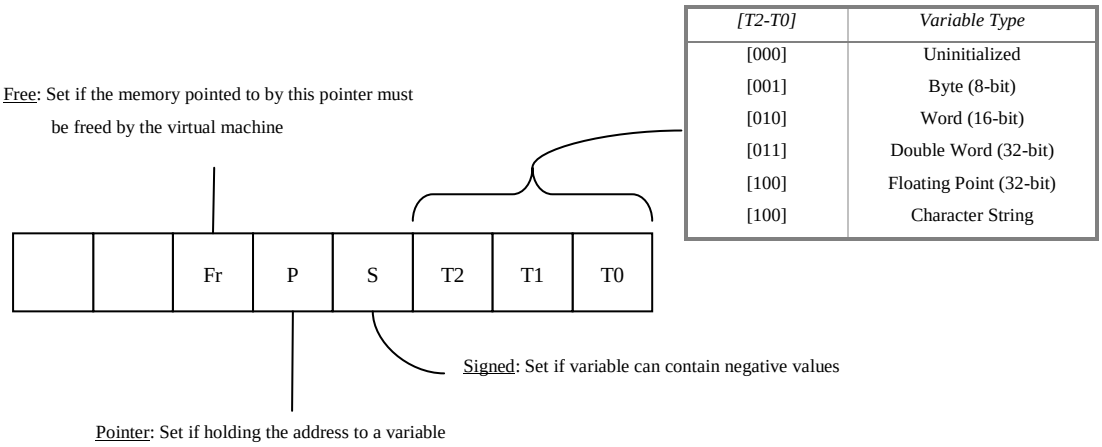
4.5.1 STACK

A memory stack is a powerful tool used by applications written for hardware architectures. A stack is useful for temporarily storing values, for example during mathematical calculations or when registers are required for an operation. Therefore, a stack was also implemented for the virtual machine due to the versatility it provides.

One of the configuration options of the virtual machine is the size of the stack. This must be defined before any application can be executed. This option can be defined manually at the start of a script, but for this work it will be calculated and set by the compiler (explained later in this chapter).

A 32-bit wide stack was selected to simplify programming instructions and to reduce type conversions between variables. In addition to this primary-stack, an extra 8-bit stack was also implemented. This 8-bit stack is the type-stack and is used to store type-definition information about a corresponding variable in the primary-stack. This 8-bit stack will henceforth be referred to as the type-stack and stores type-identifying information about its corresponding index in the primary-stack.

Figure 4-8 summarizes the format of a type-stack entry.



[T2-T0]	*Variable Type*
[000]	Uninitialized
[001]	Byte (8-bit)
[010]	Word (16-bit)
[011]	Double Word (32-bit)
[100]	Floating Point (32-bit)
[100]	Character String

FIGURE 4-8: TYPE-STACK BIT FORMAT

The first three bits (T0-T2) identify the type of variable placed on the primary-stack. An additional bit (S) specifies whether the variable is signed[7] or unsigned[8]. If the pointer bit (P) is set, the primary-stack contains a memory pointer to the value, instead of the actual value.

The character string variable type implicitly requires the pointer bit to be set, because it must point to a block of memory containing the string of several bytes. Alternatively, if the pointer bit is not set, the value is interpreted as the index to a string variable (discussed in the next section).

The free bit (Fr) indicates that when this variable is removed from the stack, the memory associated with it must be returned to the system (freed). An example, of when this bit will be used, is when the virtual machine allocates system memory to hold a temporary string variable. When this variable is no longer needed, the virtual machine will see the bit and notify the system that the memory block must be freed.

[7] Signed values can be positive as well as negative. This however halves the maximum value that can be represented. For example, an 8-bit variable can hold 256 values (0 to 255). However, if signed values can be represented, the range of values become: -128 to 127.
[8] Unsigned values can only represent positive values. For example an 8-bit variable can only represent values of 0 to 255.

The stack is useful for temporarily storing values, for example, when mathematical calculations are performed, but only the value at the top of the stack can be accessed directly.

4.5.2 VARIABLES

Variables are directly accessible storage units that can hold a value. Indexed variable arrays are used to store variables in this implementation of the VM. An array for each type of variable (bytes, words, double words, strings, etc.) is allocated in memory at the start of the user application, each according to a precalculated size. Any variable can then be accessed directly with an index and type.

User variables are defined by the user during initial configuration. These variables have meaningful names (for example, water level) that are translated to an index and data type when the user application is compiled.

To maximize execution speed and limit the amount of memory space occupied by the variable arrays, the index is chosen to be eight bits wide. Thus a total of 256 variables of each type can be used in a user application (note that for systems with greater resources available, this number can easily be increased).

The different variable types are listed below:

- Byte variables (8-bit);
- Signed byte variables (7-bit + sign);
- Word variables (16-bit);
- Signed word variables (15-bit + sign);
- Double word variables (32-bit);
- Signed double word variables (31-bit + sign);
- Signed floating point variables (32-bit);
- Character string variables (system pointer to block of memory holding the string).

String variables are unique since they hold a memory pointer to the actual string of characters. Each string variable has an additional flag to indicate whether it holds the address to memory that must be freed.

4.5.3 SCRIPT

Traditionally, scripts are typed in human readable form and executed directly by parsing this text file line by line during execution. Text parsing, however, requires significant processing before it can be executed. Thus, in order to improve performance (for this embedded application) the script is pre-parsed into an executable byte code before it is downloaded onto the PLC. This byte code is synonymous with machine language of hardware architectures. Each byte sequence informs the virtual machine which action to perform next.

Some of these instructions require additional data to perform their function. For example, a jump instruction needs the location to jump to. This leads to two possible options:

1. Make every instruction the same length (for example, 32-bit wide);
2. Allow instructions to differ in length, depending on the amount of data they require (for example, 8-bit instructions with optional data).

The first option greatly simplifies execution of the script as well as compilation. The reason for this is that the exact offset of the next instruction is always known beforehand. This allows code branching instructions to be independent of the type of instructions and only dependent on the number of instructions between source and destination. Figure 4-9 illustrates this along with the alternative (variable length instructions).

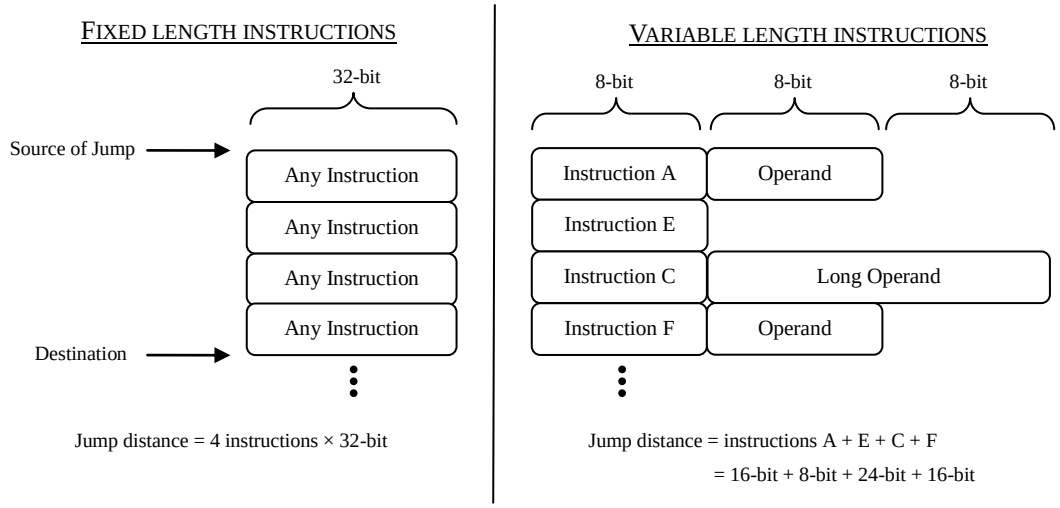


FIGURE 4-9: CODE BRANCHING COMPLEXITY OF VARIABLE LENGTH INSTRUCTIONS

This approach, however, comes at the expense of greater memory usage. Because memory is a limited resource, it is better to employ a method that uses less memory space, even if this increases the complexity of the virtual machine.

This leads to the second approach where a short instruction code is assigned with an optional data operand for some instructions. The only disadvantage is a more complex compiling algorithm and initial PLC firmware development (no runtime implications are incurred). Thus, for this project, it is decided to implement an 8-bit instruction set (up to 256 different instructions) with a variable length operand for some instructions.

For a complete list of these byte codes, please refer to Appendix C.

4.5.4 INTERPRETER

The interpreter is similar to a hardware processor. It is responsible for managing registers and executing program instructions. In order to achieve this, the virtual machine requires certain core registers. (Note that, because this is a virtual machine, a register does not correspond to an actual hardware register, but instead a software variable). These registers are:

- Instruction Pointer (IP): Points to the next instruction in the script to be executed;
- Stack Pointer (SP): Points to the next empty location on the stack;
- Active Flag: Set if virtual machine is active, clear if suspended.

The implementation of the interpreter is dependent on whether the operating system provides multiprogramming support. Without this feature, the virtual machine needs to return control to the operating system after the execution of every instruction. This is illustrated in Figure 4-10 on the next page:

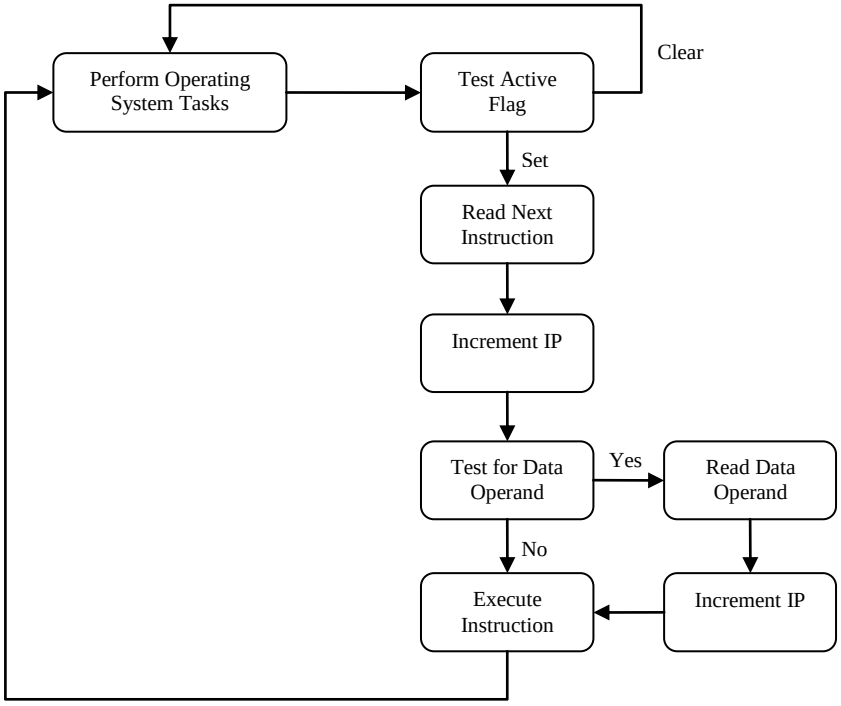


FIGURE 4-10: INTERPRETER FLOW (WITHOUT MULTIPROGRAMMING)

The interpreter starts by testing whether the virtual machine is active. If not, it returns control to the operating system. Otherwise, it reads the next instruction from the script and increments the instruction pointer (IP). If that instruction requires an additional data operand, it is also read from the script and the instruction pointer is updated. The instruction is then executed and control is returned to the operating system.

If multiprogramming is supported (as is the case for this project), a simpler approach can be followed by allowing the virtual machine to execute in its own enclosed thread. The operating system will periodically take control and perform any required tasks. Afterwards control will automatically be returned to the virtual machine. This allows the virtual machine to disregard any operating system requirements and thus simplifies its development. Figure 4-11 on the next page illustrates this approach.

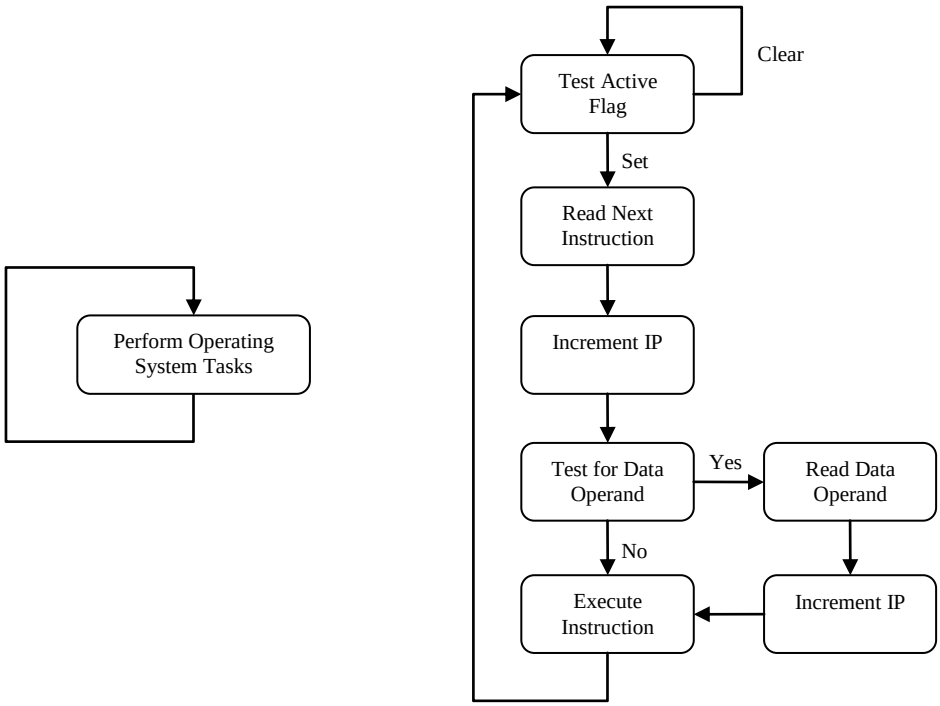


FIGURE 4-11: INTERPRETER FLOW (WITH MULTIPROGRAMMING)

4.5.5 SYSTEM CALL HANDLER

The system call handler can be thought of as the bridge between the virtual machine and the actual underlying hardware system. For this reason, it is also the one component of the virtual machine that is system dependent. This entails that, for the virtual machine to function on another hardware platform, this module must be rewritten for the new system. This implicitly improves the portability of the other VM modules due to the abstraction layer provided by the system call handler.

The system call handler is comprised of two parts. The first part consists of several handlers (one for each system call). Each handler is represented by a software function that, when called by the virtual machine, performs a desired system action. For example, there is a handler for setting a digital output and another handler to clear it. This is similar to a set of high-level driver functions that form an abstraction layer between the VM and the underlying operating system.

The second part is a list or array that contains all the addresses of the different handlers (function pointers, effectively). This was done so that the virtual machine can instantly call the correct handler from an index[9]. For example, the C code to implement a system function call is simply:

```
HandlerPointers[IndexOfSystemCall]();
```

The alternative would require processing of the index to determine which handler to call. This will not only incur a performance loss, but leads to bulky source code, which makes the addition of new system calls difficult. An example in C code could be:

```
if(IndexOfSystemCall == 0)
{
   HandlerOne();
}
else if(IndexOfSystemCall == 1)
{
   HandlerTwo();
}
.
.
.
```

Because this module only serves as a bridge, it is required that the functionality be implemented on the actual hardware platform as well. This is necessary to ensure the VM remains portable, since these functions are application and hardware specific. For example, if the virtual machine needs to change the state of a digital output to “high”, it will execute a system call instruction for the “Set Digital Output” system function. This will initiate the corresponding handler which calls the actual firmware function to set the output high.

[9] Note that this index is zero based, because in the c programming language the first element in an array is denoted by 0.

For system functions that require data, the script can push it onto the stack prior to calling the handler. The handler will then take this value from the stack and pass it to the actual system function as a parameter. An example of this is shown in Figure 4-12, where the third digital output is set to “high”. For clarity, the operational flow is numbered sequentially in the order of occurrence.

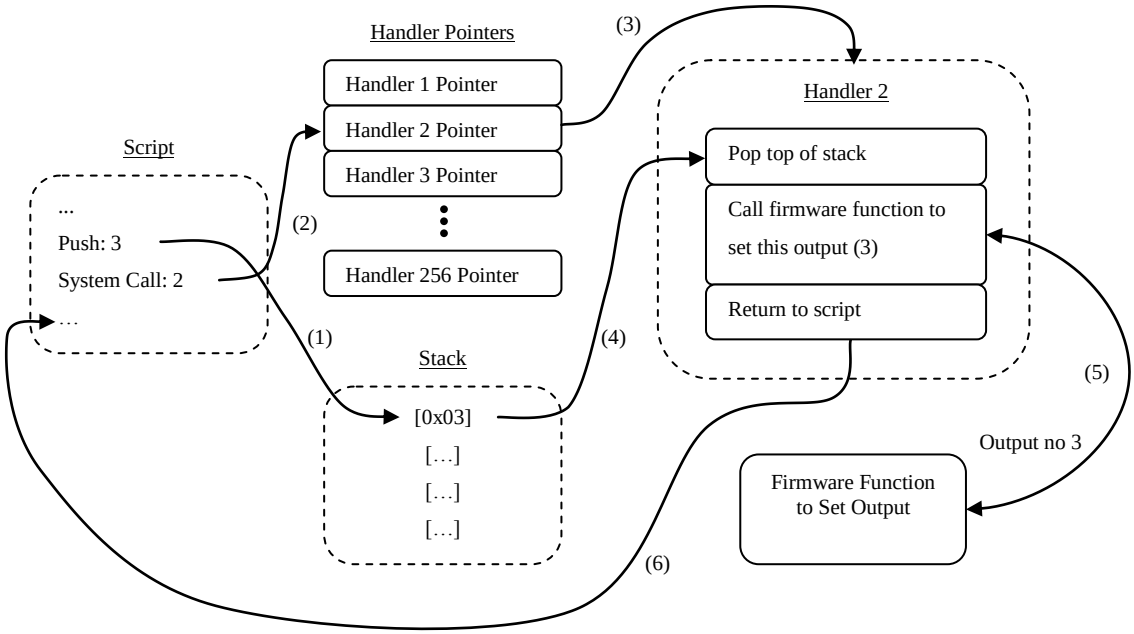


FIGURE 4-12: PARAMETER PASSING TO SYSTEM CALLS

Similarly, the stack is also used for system calls that return a value to the VM. An example that requests the time of day in seconds is shown in the following figure. Note that the time value is placed on the stack and is available for processing after returning to the script.

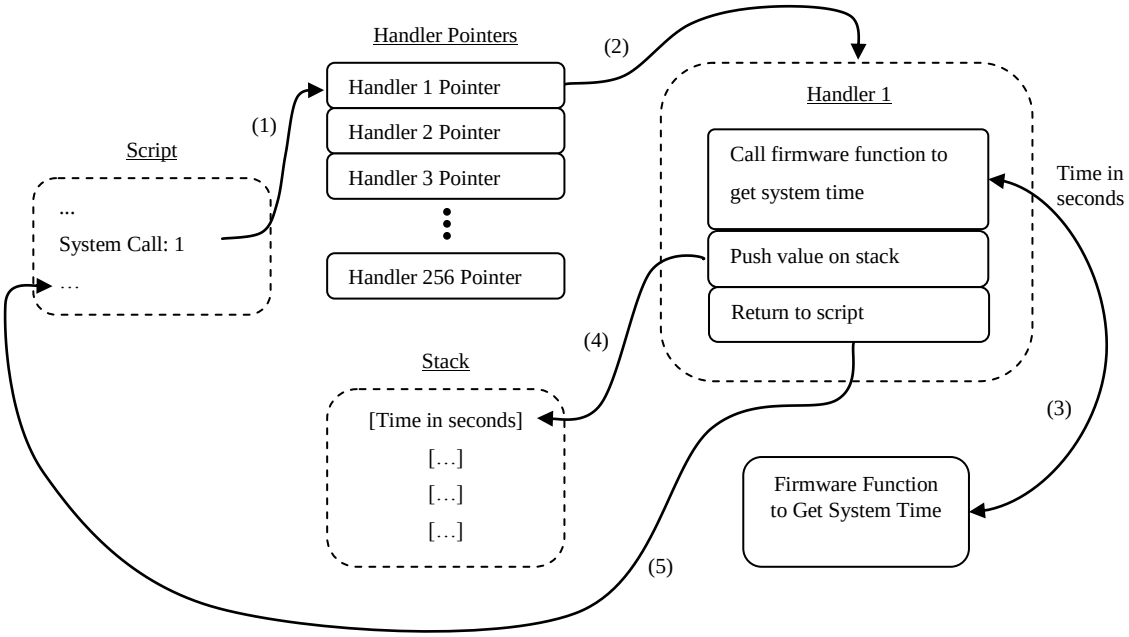


FIGURE 4-13: PARAMETER PASSING FROM SYSTEM CALLS

For a complete list of implemented system calls, please refer to Appendix D.

4.6 SUMMARY

This chapter explained the detailed design of the project elements, which included the design of the PLC computer software *(including high-level programming language and compiler but excluding GUI)* as well as the design and programming of the PLC firmware *(including virtual machine, VM instruction set, interpreter, and scripts).*

The elements that were discussed in the PLC computer software design are:

- Development of high-level event-based programming language *(if-then-else clauses)*;
- Compilation from high-level to low-level PLC code;
- Connection establishment between computer and PLC;
- Packet protocol with error correction for reliable communication between computer and PLC;
- Command-based protocol for controlling and programming the PLC via the PLC computer software.

The elements that were discussed in the PLC firmware design are:

- Operating system selection and implementation;
- Connection handler to maintain connectivity between the PLC and PLC computer software;
- Handling of commands received from the PLC computer software;
- Virtual machine with its own complete VM instruction set;
- Interpreter for execution of user-programmed applications;
- Power-up handler to initialize PLC to a predefined state.

It is concluded that the design of the interpreter, comprising the virtual machine, interpreter, PLC computer software, and event-based programming language, has been completed and all input and design requirements for this system have been addressed.

CHAPTER 5
COMPARISON OF OPERATING SYSTEMS

5.1 INTRODUCTION

The operating system is critical for this implementation of a PLC since it determines the responsiveness of the system. Thus, it was necessary to perform an experiment to test and evaluate the performance of two different operating systems.

As a first option, Telit has provided a Linux platform for firmware development on their module. This is a very powerful operating system that provides almost all of the advanced OS features, even including virtual memory [10][16]. The performance cost incurred by these features (most of which are unneeded for this project) and the fact that Linux is not a hard real-time operating system, is unknown.

Due to these unknown performance implications, it was decided to implement a low-end alternative, in order to perform a comparative study. This led to the second option, FreeRTOS, which is a low-end embedded operating system. FreeRTOS provides hard real-time functionality, as well as the required OS features for this work (such as, multiprogramming support) [9].

An experiment was set up to determine the performance difference between uCLinux and FreeRTOS – this experiment is described in this chapter.

5.2 PURPOSE

The purpose of this experiment is to determine which operating system provides the best performance for this project.

5.3 METHOD

The method is based on a qualitative comparison of two operating systems, namely uCLinux and FreeRTOS. Each operating system implements the same virtual machine and executes the same script. The time taken to execute this script will be used as the measurement for comparison. (Note that as a result, it was necessary to implement the VM on both uCLinux and FreeRTOS)

The experiment will require the following steps:

- Implement a working FreeRTOS system;
- Modify the Linux based firmware to function within FreeRTOS;
- Write a basic script that implements all or most of the core functionality;
- Execute script and measure performance.

5.4 EXPERIMENTAL SETUP

Due to time limitations as well as Telit restrictions which prevent porting of FreeRTOS to their module, the comparison was done on two different processors.

1. The ARM9 (AT91SAM9260) processor running uCLinux at 100MIPS;
2. An ARM7 processor running a FreeRTOS port at 40MIPS.

The difference in processors will affect the results. However, since both processor cores are ARM-based and the processing speed is known, normalization of the measurements can be done afterwards (within known accuracy limits). Thus, any time-based measurements can be scaled proportionally to provide a comparable result.

Additionally, since FreeRTOS is run on the smaller core (potentially faster operating system on a slow core vs. the complex operating system on a fast core), there exists the possibility that the FreeRTOS implementation may come close to or even surpass the uCLinux implementation in speed. Irrespective of the result, valuable insight can be gained from the comparison.

The ARM7 was chosen because an AT91SAM7X-EK development board from Atmel was available [20]. An additional reason for choosing the ARM7 is that an existing FreeRTOS port exists for this processor (more accurately, for this specific development board) [9]. This eliminates the need to implement a functional FreeRTOS platform.

The firmware port from Linux to FreeRTOS was a simple task due to the design of the virtual machine abstraction. FreeRTOS also provides the same OS features used by the Linux implementation (for example, multiprogramming).

The script used for the comparison is shown and explained in Figure 5-1. A variable is used to count program cycles and can be used to time the speed of execution. After one million program cycles, an output connected to an LED is switched on. This time (from start till LED switches on) can be measured manually in seconds. Because of the large number of cycles executed within this period, the average program cycle time can be calculated with microsecond accuracy. This script also uses several virtual machine features, namely:

- Digital I/O with buffering;
- Code branching (jumps);
- Stack operations;
- Variables;
- Constants;
- Math operators;
- System Calls.

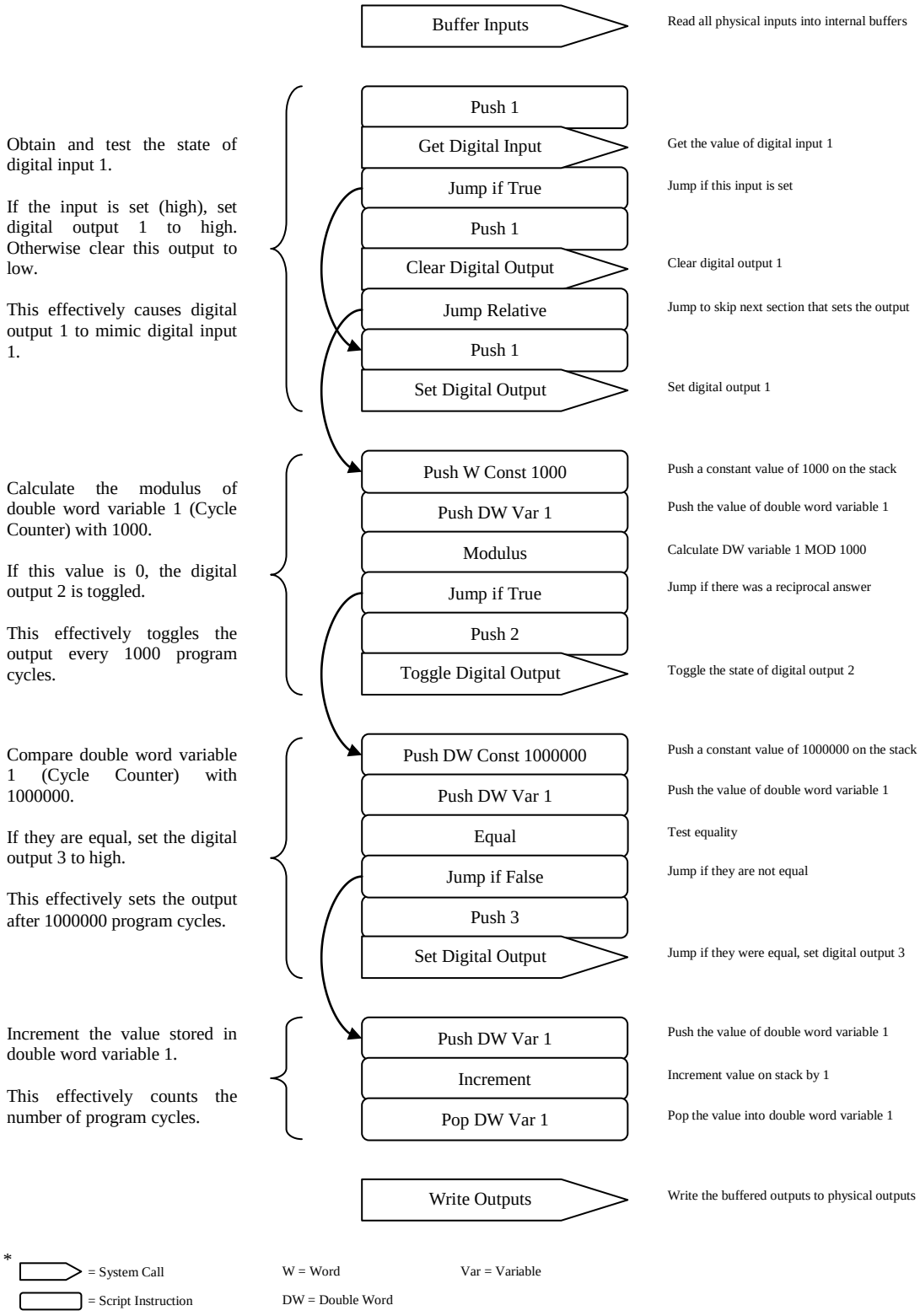


FIGURE 5-1: OPERATING SYSTEM COMPARISON SCRIPT

5.5 RESULTS

The execution time measurements are summarised in the following table.

TABLE 5: OPERATING SYSTEM EXPERIMENT RESULTS

Operating System	Processor Speed (MIPS)	Time for one million program cycles	Average time per program cycle	Average time per program cycle (Proportionally corrected)
uCLinux	100	23s	23µs	23µs
FreeRTOS	40	17s	17µs	6.8µs

5.6 EVALUATION AND CONCLUSION

From the results, it is clear that the FreeRTOS implementation executes the script significantly faster than the Linux alternative. This is evident even before a proportional correction of the results. This can most likely be attributed to the extra processing overhead required by the advanced features of Linux.

Additionally, FreeRTOS provided all required operating system functionality for this project. It also proved to be easy to use with a simple interface to directly and safely access the hardware. Linux, on the other hand, has a complicated and strict hardware driver system. This significantly prolonged development time as well as reducing the frequency with which hardware interrupts can be serviced.

Ultimately, it can be concluded that FreeRTOS is the best alternative for this project.

At this stage, it is necessary to give recognition to work that was done by van der Merwe [21], where work was done on the real-time characteristics (or lack thereof) of Linux for embedded systems. The work done in van der Merwe's research was done in parallel to the research done in this study and confirms that Linux does not support real-time applications. This confirms our selection of an operating system, namely FreeRTOS, where the firmware programmer has control over real-time operations.

CONCLUSION

The first section of the conclusion provides an overview of the system that was developed in this work and provides all project objectives that have been addressed.

A fully functional PLC system was developed in this project, as requested by the client. The PLC firmware is based on a *virtual machine* that executes a sequence of low-level instructions from a *script*. These low-level instructions are chained to give flexible high-level PLC functionality. The script itself is interpreted by a *real-time interpreter*, providing a platform for user-programmed PLC applications.

The actual programming of the PLC is done with the use of *specialised computer software*. This software allows the programmer to define the PLC operations with *event-action* pairs in the intuitively understandable human format of *if-then-else* clauses. This greatly improves the usability of the PLC for embedded projects. The simplified programming language and user-friendly computer software can significantly reduce time-to-market of products, leading to *rapid product development* due to this PLC system.

The specialised computer software communicates with the PLC using a *packet protocol* and a *command-based message structure*. The packet encapsulates the data to be sent *(commands)* in a data format with all required fields for transmission and a 16-bit CRC for error checking to increase the reliability of the data. Once received by the PLC, these commands are executed and the PLC performs the requested action.

An *operating system* was implemented on the PLC firmware, in order to provide a stable and reliable interface for the virtual machine to access the underlying PLC hardware and system functionality. This also improved overall system robustness and portability. A comparative experiment *(*between *uCLinux* and *FreeRTOS)* was performed to find the operating system that provides optimal performance for this system. The results show that FreeRTOS, with its simplistic design and ease-of-use, provides superior performance.

With the above mentioned system, all project objectives have been addressed. These objectives are listed again for convenience and consist of:

- A computer interface that is easy to use with rule-based logic functions and configuration options *(implemented by the specialised computer software and if-then-else event-based programming language)*;
- A reliable communication protocol to interface the PLC to the computer software *(implemented by the packet protocol and command-based message system)*;
- An interpreter to execute all user-programmed applications on the PLC platform *(addressed by the interpreter as a whole, including virtual machine, VM instruction set, and the scripts)*;
- An embedded operating system that is recognized, stable, and that provides real-time functionality *(both uCLinux and FreeRTOS have been implemented and an experiment was conducted which showed that FreeRTOS provides superior performance for this system)*;
- A functional PLC module that integrates all elements of the system *(the complete PLC system, consisting of all above mentioned elements, was implemented)*.

The following section shows that the requirements for this research and development project were all met in the design and implementation of an interpreter for a generic PLC. The *input requirements* for the system are shown again for convenience, followed by a description of how it was addressed:

- GSM-communication (SMS, GPRS);
- Analog signal measurement;
- Digital input measurement;
- Digital output manipulation;
- Pulse counting;
- Event timing;
- Mathematical manipulation of data;
- Programmer definable variables for data handling;
- Windows-based PLC configuration and programming;
- Intuitive and logical programming language and interface;
- General manipulation of data and outputs using rule-based events;

- Connectivity between PLC and computer via RS-232, USB, or GPRS;
- Remote monitoring of PLC's I/O and data;
- Portable and robust code on the PLC module itself.

Support for *analog inputs*, *digital inputs* and *outputs*, and *variables* was fully implemented for the PLC system and the design is explained in detail in this document (see Chapters 3, Chapter 4 and Appendix A). PLC functions for *GSM communication services (SMS and GPRS)*, *pulse counters*, and *timers*, are also supported by the interpreter and were fully implemented. These features, however, are not discussed in detail because their interaction with the interpreter is for all practical purposes identical to that of the I/O and variables (apart from the fact that different events and actions are associated, the actual virtual machine remains unchanged).

A *Windows-based computer application* was programmed within the Microsoft .NET platform in C#, in order to provide the PLC programmer with a usable interface to configure and program the PLC. The actual programming of the PLC is done by defining *events* with a *rule-based* language based on an *intuitive programming language* consisting of *if-then-else clauses*.

The Windows-based computer application, in conjunction with the PLC hardware and firmware, also provides support to establish a *connection between the PLC and computer software* via a *serial port (RS-232)*, *USB*, or *GPRS network*. This connection is used to program and configure the PLC and also allows for *remote monitoring of the PLC*.

During the development of the system several *general design requirements* were taken into account to improve *portability*, *usability*, and *robustness*. The design requirements were derived from the above mentioned *input requirements*, as well as requirements that have resulted from years of refinement and tried and tested principles as found in existing systems (refer to the *literature study* in Chapter 2). The principles followed in this design, to address these design requirements, are discussed in the final section of this chapter.

In order to improve the portability of the system, the following was implemented:

- An operating system *(specifically uCLinux and FreeRTOS)* to provide the system with an abstraction layer between the specific PLC hardware architecture and the firmware;
- A virtual machine to provide an additional abstraction layer between the operating system and the virtual machine *(this allows the same virtual machine to be implemented on any architecture as long as the operating system remains the same, otherwise minor adjustments are required with regards to the operating systems)*;
- A generic scripting language that runs within the virtual machine *(this allows the same scripts to be executed on any platform that implements the virtual machine).*

In order to make the product usable, the following was implemented:

- A user-friendly computer application *(note that the graphical user interface does not form part of this work and is not discussed in this document)*;
- A flexible high-level, event-based programming construct consisting of sequential condition-action pairs *(this aims to improve the usability by only allowing one main execution flow, effectively improving the deterministic nature of the application and reducing the number of aspects to consider simultaneously while programming);*
- An intuitive human-understandable programming language based on *if-then-else clauses*.
- Support to evaluate events at specific times or fixed periodic intervals, providing the programmer with greater control over when operations occur.

To improve the robustness of the interpreter, the following was implemented:

- A recognized and stable operating system *(specifically uCLinux and FreeRTOS)* to provide a reliable interface between hardware and firmware as well as required OS features, such as multiprogramming;
- A virtual machine that provides additional robustness in that only *recognized* and *tested* instructions are allowed *(this also restricts direct access to low-level functionality via protected and safe instructions)*;

- A single code path or sequential execution *(this improves the deterministic nature of the applications, eliminating unexpected results that can occur with multiple code paths)*;
- Independent low-level interpretation of events *(this allows the compiler to focus on each event separately, eliminating the possibility of any events breaking the execution of any other event)*;
- Input-synchronization by only reading input ports once at the onset of a program cycle. External changes to inputs during the event evaluation period are recorded but only evaluated at the start of the next cycle *(this causes the entire PLC application to be executed with the same input states, effectively eliminating unexpected results which could occur if input states changed in the middle of an event or between matching events)*;
- Output-synchronization by only allowing changes to output ports to take effect at the end of each program cycle.

All input and design requirements, as baselined for this system, have been met. It can thus be concluded that all requirements have been met for the development of a *usable, real-time interpreter for rapid product development*.

APPENDIX

APPENDIX A: PLC PROGRAMMING INTERFACES

Available programming conditions and actions are predefined and associated with a specific interface, where an interface is any part of the PLC (logical or physical) that can form a condition or perform an action. For example, a digital input is an interface that can be used to form a condition (if digital input is high), but it cannot perform an action. A digital output can form a condition (if digital output is low), as well as perform an action (digital output is set to high).

An overview of the interface structure is shown in the following figure.

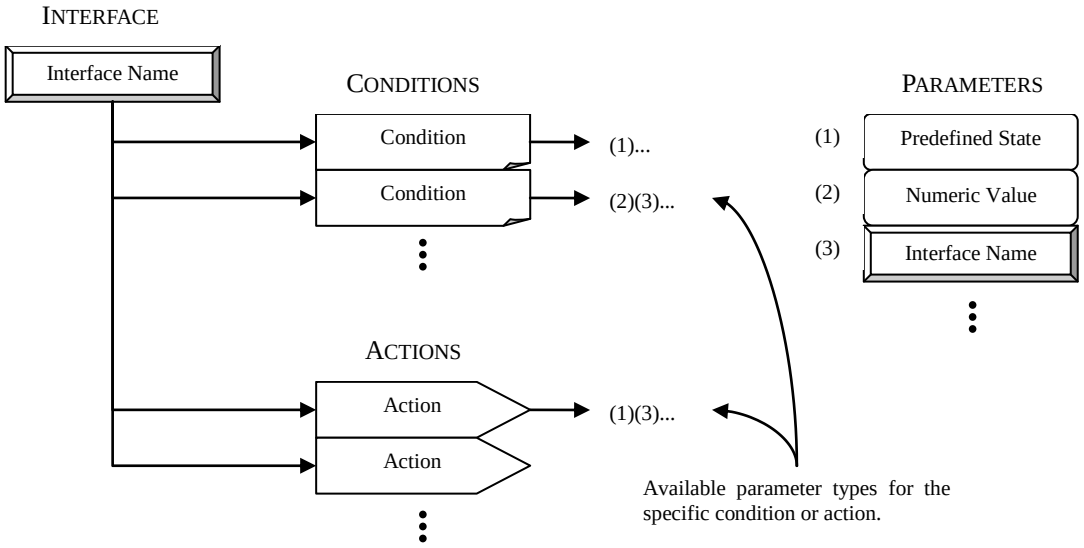


FIGURE A-1: INTERFACE STRUCTURE

Every interface can provide a number of conditions and actions. Certain of these conditions and actions may require a parameter to complete its meaning. For example, a digital output is set to [HIGH or LOW]. The complete list of available parameters is listed per interface and numbered. These numbers are listed beside each condition or action to indicate that it is applicable. Note that interfaces can also be used as parameters if they conform to the desired type. For example, a digital output can be set to the current state of a digital input. Also, numeric values can contain mathematical expressions with several parameters.

Interfaces implemented and relevant to this work are listed below.

DIGITAL INPUT

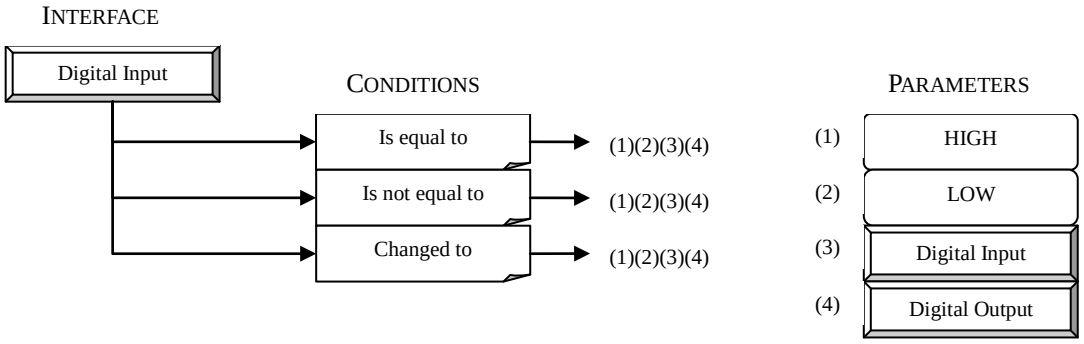


FIGURE A-2: INTERFACE: DIGITAL INPUT

The digital input interface, allows access to a physical digital input. No actions are provided, but conditions to test the current input state against other digital ports or fixed levels are provided. An extra condition is provided that tests whether the state of the input has changed since last tested.

DIGITAL OUTPUT

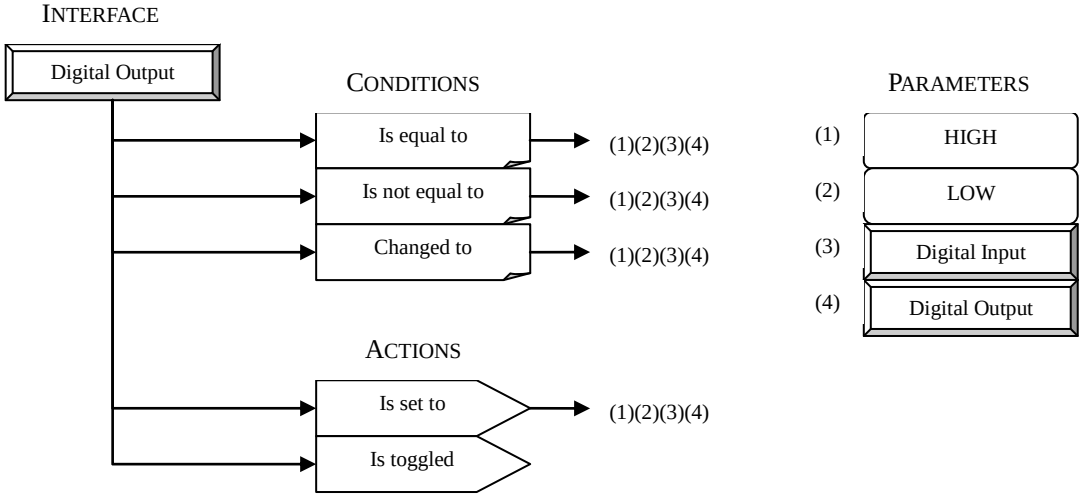


FIGURE A-3: INTERFACE: DIGITAL OUTPUT

The digital output interface, allows access to a physical digital output. The same conditions are provided as for the digital input interface. Actions are also provided to toggle the output or set it equal to other digital ports or fixed levels.

ANALOG INPUT

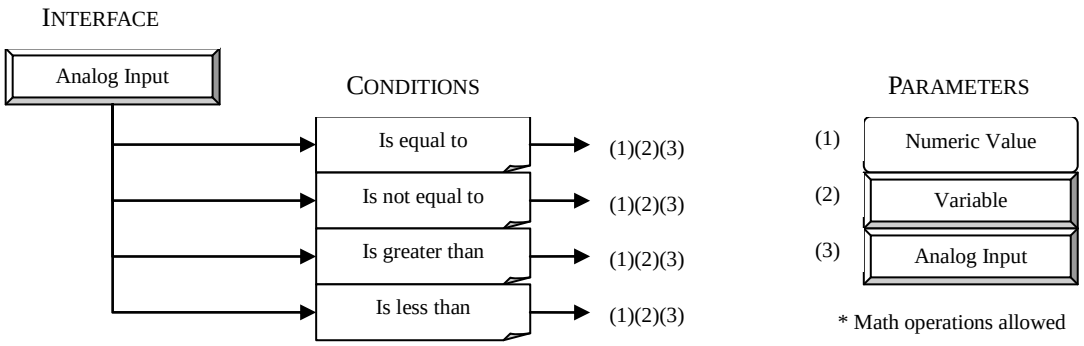


FIGURE A-4: INTERFACE: ANALOG INPUT

The analog input is similar to the digital input, except that a range of values are supported. For this reason, the interface can be compared to numerical values, variables or other analog interfaces.

VARIABLE

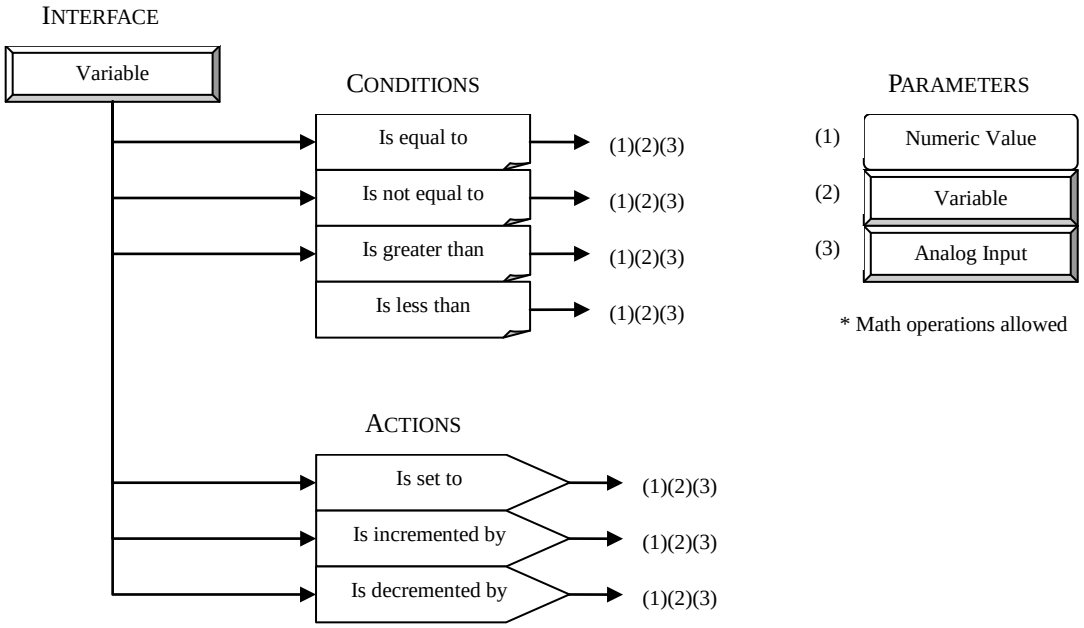


FIGURE A-5: INTERFACE: VARIABLE

The variable is a logical interface to access placeholders for numeric values. Conditions are provided for comparison between other variables or compatible interfaces. A few actions are also provided to modify the value held by the variable. Variables are frequently used in mathematical calculations, by chaining several parameters with math operators (+ - × ÷ etc.).

APPENDIX B: COMPUTER ⇔ PLC COMMANDS

The following figure shows the three different command formats.

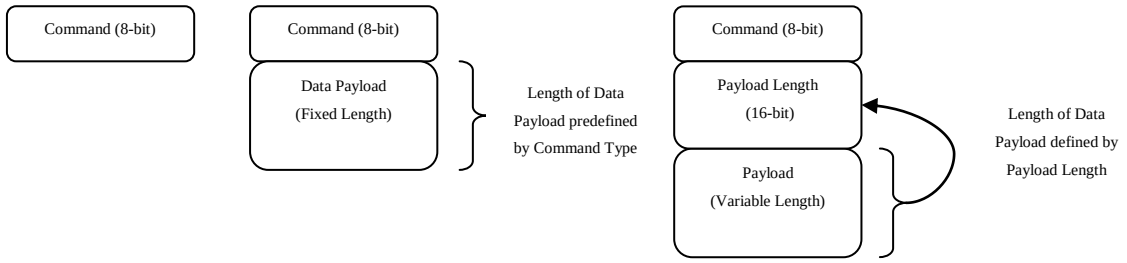


FIGURE A-6: COMMAND AND COMMAND-WITH-PAYLOAD FORMATS

The commands are primarily used to signal requests from the host (computer) to the device (PLC). Note that these commands are also used as return codes from the device to the host. This is the case if data is requested from the device or to signal success or failure of the request.

Commands implemented in this project are listed in tabular form:

TEST CONNECTION

Name	Test Connection
Command Code	0x00
Description	Sent to the PLC to test the connection. If the computer receives the response, it is assumed the connection is active.
Data Payload (Computer to PLC):	None
Data Payload (PLC to Computer):	None
Example	Computer → 0x00 → PLC (Request connection test) Computer ← 0x00 ← PLC (Reply indicates success)

START VIRTUAL MACHINE

Name	Start Virtual Machine
Command Code	0x01
Description	Enables the virtual machine and starts execution of any available script.

Data Payload (Computer to PLC):	Fixed Payload:
	• 1 Byte
	  ○ 0x00 = Start from the beginning of the script
	  ○ 0x01 = Continue from last stop

Data Payload (PLC to Computer):	Fixed Payload:
	• 1 Byte
	  ○ 0x00 = Success
	  ○ 0xFF = Failure (Script not valid)

Example	Computer → 0x01 0x01 → PLC (Request VM start and continue) Computer ← 0x01 0x00 ← PLC (Reply with success)

STOP VIRTUAL MACHINE

Name	Stop Virtual Machine
Command Code	0x02
Description	Disables the virtual machine and stops execution of the script.
Data Payload (Computer to PLC):	None

Data Payload (PLC to Computer):	Fixed Payload:
	• 1 Byte
	  ○ 0x00 = Success
	  ○ 0x01 = Success but VM was already stopped

Example	Computer → 0x02 → PLC (Request VM stop) Computer ← 0x02 0x00 ← PLC (Reply with success)

PROGRAM SCRIPT

Name	Program Script
Command Code	0x03
Description	Sends a script to the virtual machine, so that it can be executed. (Note that this command automatically stops the virtual machine)
Data Payload (Computer to PLC):	Variable Payload: • 1-65535 Bytes = Script
Data Payload (PLC to Computer):	None
Example	Computer → 0x03 [Script Length] [Script Data] → PLC Computer ← 0x03 ← PLC (Reply indicates success)

VERIFY SCRIPT

Name	Verify Script
Command Code	0x04
Description	Sends a script to the virtual machine, so that it can be compared to the current script.
Data Payload (Computer to PLC):	Variable Payload: • 1-65535 Bytes = Script

Data Payload (PLC to Computer):	Fixed Payload:
	• 1 Byte
	  ○ 0x00 = Scripts match
	  ○ 0xFF = Scripts do not match

Example	Computer → 0x04 [Script Length] [Script Data] → PLC Computer ← 0x04 0x00 ← PLC (Reply with match)

SAVE SCRIPT

Name	Save Script
Command Code	0x05
Description	Request PLC to save the current active script to non-volatile memory.
Data Payload (Computer to PLC):	None

Data Payload (PLC to Computer):	Fixed Payload:
	• 1 Byte
	  ○ 0x00 = Script successfully saved
	  ○ 0xFE = Failure (Saving failed)
	  ○ 0xFF = Failure (No active script)

Example	Computer → 0x05 → PLC (Request save) Computer ← 0x05 0x00 ← PLC (Reply with success)

GET DIGITAL OUTPUT

Name	Get Digital Output
Command Code	0x06
Description	Request the state of a specific digital output.
Data Payload (Computer to PLC):	Fixed Payload: • 1 Byte = Index of digital output

Data Payload (PLC to Computer):	Fixed Payload:
	• 1 Byte
	  ○ 0x00 = Digital output is low
	  ○ 0x01 = Digital output is high
	  ○ 0xFF = Error (Invalid index)

Example	Computer → 0x06 0x02 → PLC (Request state of digital output 2) Computer ← 0x06 0x01 ← PLC (Reply indicates high state)

GET DIGITAL INPUT

Name	Get Digital Input
Command Code	0x07
Description	Request the state of a specific digital input.
Data Payload (Computer to PLC):	Fixed Payload: • 1 Byte = Index of digital input

Data Payload (PLC to Computer):	Fixed Payload:
	• 1 Byte
	  ○ 0x00 = Digital input is low
	  ○ 0x01 = Digital input is high
	  ○ 0xFF = Error (Invalid index)

Example	Computer → 0x07 0x04 → PLC (Request state of digital input 4) Computer ← 0x07 0x01 ← PLC (Reply indicates high state)

GET ANALOG INPUT

Name	Get Analog Input
Command Code	0x08
Description	Request the value of a specific analog input.
Data Payload (Computer to PLC):	Fixed Payload: • 1 Byte = Index of analog input
Data Payload (PLC to Computer):	Fixed Payload: • 1 Word = Digital value from ADC
Example	Computer → 0x08 0x03 → PLC (Request value of analog input 3) Computer ← 0x08 0x0128 ← PLC (Reply with value 296)

GET ANALOG INPUT RANGE

Name	Get Analog Input Range
Command Code	0x09
Description	Request the maximum digital value of a specific analog input. (The range is 0 to maximum)
Data Payload (Computer to PLC):	Fixed Payload: • 1 Byte = Index of analog input
Data Payload (PLC to Computer):	Fixed Payload: • 1 Word = Maximum digital value from ADC
Example	Computer → 0x09 0x03 → PLC (Request range of analog input 3) Computer ← 0x09 0x03FF ← PLC (Reply with value 1023)

APPENDIX C: VIRTUAL MACHINE INSTRUCTION SET

Traditionally, scripts are typed in human-readable form and executed directly by parsing this text file line by line during execution. Text parsing, however, requires significant processing before it can be executed. Thus, in order to improve performance (for this embedded application) the script is pre-parsed into an executable byte code before it is downloaded onto the PLC. This byte code is synonymous with machine language of hardware architectures. Each byte sequence informs the virtual machine which action to perform next.

Because of the large number of byte codes implemented in this application, they will be explained in groups of similar functionality.

INEQUALITY

These instructions are used to compare the top two values on the stack with each other. These entries are then replaced with the Boolean result.

Byte Code	Name	Description
0x01	Greater	Performs (Top > 2nd From Top) operation on the top 2 stack entries and replaces them with the result
0x02	Greater or Equal	Performs (Top ≥ 2nd From Top) operation on the top 2 stack entries and replaces them with the result
0x03	Less	Performs (Top < 2nd From Top) operation on the top 2 stack entries and replaces them with the result
0x04	Less or Equal	Performs (Top ≤ 2nd From Top) operation on the top 2 stack entries and replaces them with the result
0x05	Equal	Performs (Top = 2nd From Top) operation on the top 2 stack entries and replaces them with the result

LOGIC OPERATORS

Logic operators are used to concatenate Boolean values (true or false) on the stack, in order to evaluate complex logical expressions. It is important to perform the operations in the correct order. Otherwise, different (and incorrect) results can be obtained, for example, [A OR B AND C] can be interpreted as both [(A OR B) AND C] and [A OR (B AND C)].

Note that only the top two entries are affected by these operators. For the "Not" operator, only the top entry is affected.

Byte Code	Name	Description
0x06	And	Performs logic AND operation on the top 2 stack entries and replaces them with the result
0x07	Or	Performs logic OR operation on the top 2 stack entries and replaces them with the result
0x08	Not	Logic inverts the top of the stack (true → false and false → true)

PROGRAM BRANCHING

These instructions are used to change the flow of program execution, also called a jump. An operand indicates a relative distance to move the program counter. A value of 0 would cause no jump to occur and execution will continue to the next instruction. This value is also signed, giving the ability to jump backwards. The basic jump has a byte operand and can move a distance from -128 to 127. A far jump version is also implemented with a word operand, allowing a distance of -32768 to 32767.

Some jumps are conditional and depend on the top entry of the stack. After these jumps, the top of the stack is automatically popped. There is also an absolute jump that changes the program counter to a specified value. This allows a jump to anywhere in the script.

Byte Code	Name	Operand	Description
0x09	Jump If True	Byte = Distance	If the top of the stack is true, the program counter (PC) is displaced by [Distance] (-128 to 127).
0x0A	Jump If False	Byte = Distance	If the top of the stack is false, the program counter (PC) is displaced by [Distance] (-128 to 127).
0x0B	Jump Far If True	Word = Distance	If the top of the stack is true, the program counter (PC) is displaced by [Distance] (-32768 to 32767).
0x0C	Jump Far If False	Word = Distance	If the top of the stack is false, the program counter (PC) is displaced by [Distance] (-32768 to 32767).
0x0D	Jump Relative	Byte = Distance	The program counter (PC) is displaced by [Distance] (-128 to 127).
0x0E	Jump Far Relative	Word = Distance	The program counter (PC) is displaced by [Distance] (-32768 to 32767).
0x0F	Jump Absolute	Double Word = Absolute Offset	The program counter (PC) is set equal to [Absolute Offset].

STACK

These instructions manipulate the virtual machine stack. There are three subcategories, namely generic, constant and variable instructions.

STACK (GENERIC)

Byte Code	Name	Description
0x10	Pop	Pop and discard the top to the stack.
0x11	Push Top	The top of the stack is pushed on the stack again (duplicate entry).

STACK (CONSTANTS)

These instructions push a constant value on the stack.

Byte Code	Name	Operand	Description
0x12	Push Byte	Byte = Value	Push a constant byte (signed) value onto the stack.
0x13	Push Unsigned Byte	Byte = Value	Push a constant byte (unsigned) value onto the stack.
0x14	Push Word	Word = Value	Push a constant word (signed) value onto the stack.
0x15	Push Unsigned Word	Word = Value	Push a constant word (unsigned) value onto the stack.
0x16	Push Double Word	Double Word = Value	Push a constant double word (signed) value onto the stack.
0x17	Push Unsigned Double Word	Double Word = Value	Push a constant double word (unsigned) value onto the stack.
0x18	Push Float	Float = Value	Push a constant float (signed) value onto the stack.

STACK (VARIABLES)

These instructions push the current value of a variable onto the stack.

Byte Code	Name	Operand	Description
0x19	Push Byte Variable	Byte = Variable Index	Push the value of byte variable with index = [Variable Index] on the stack.
0x1A	Push Unsigned Byte Variable	Byte = Variable Index	Push the value of unsigned byte variable with index = [Variable Index] on the stack.
0x1B	Push Word Variable	Byte = Variable Index	Push the value of word variable with index = [Variable Index] on the stack.
0x1C	Push Unsigned Word Variable	Byte = Variable Index	Push the value of unsigned word variable with index = [Variable Index] on the stack.
0x1D	Push Double Word Variable	Byte = Variable Index	Push the value of double word variable with index = [Variable Index] on the stack.
0x1E	Push Unsigned Double Word Variable	Byte = Variable Index	Push the value of unsigned double word variable with index = [Variable Index] on the stack.
0x1F	Push Float Variable	Byte = Variable Index	Push the value of float variable with index = [Variable Index] on the stack.

These instructions pop the top of the stack into a variable.

Byte Code	Name	Operand	Description
0x20	Pop Byte Variable	Byte = Variable Index	Pop the stack into byte variable with index = [Variable Index].
0x21	Pop Unsigned Byte Variable	Byte = Variable Index	Pop the stack into unsigned byte variable with index = [Variable Index].
0x22	Pop Word Variable	Byte = Variable Index	Pop the stack into word variable with index = [Variable Index].
0x23	Pop Unsigned Word Variable	Byte = Variable Index	Pop the stack into unsigned word variable with index = [Variable Index].
0x24	Pop Double Word Variable	Byte = Variable Index	Pop the stack into double word variable with index = [Variable Index].
0x25	Pop Unsigned Double Word Variable	Byte = Variable Index	Pop the stack into unsigned double word variable with index = [Variable Index].
0x26	Pop Float Variable	Byte = Variable Index	Pop the stack into float variable with index = [Variable Index].

TYPECASTING

These instructions explicitly cast the entry on top of the stack to a new type. If the previous type is larger than the requested type, the value is truncated to fit. For example, the word 0x1234 casted to a byte will be 0x23. Floating point values lose their fractional part. For example, 1234.9876 would become 1234 (note that it is not 1235, because the fraction is not rounded, but just discarded).

Byte Code	Name	Description
0x27	Cast to Boolean	Changes the top of the stack to a Boolean. $\begin{Bmatrix} Top = 0 \rightarrow false \\ Top \neq 0 \rightarrow true \end{Bmatrix}$
0x28	Cast to Byte	Changes the top of the stack to a byte (signed).
0x29	Cast to Unsigned Byte	Changes the top of the stack to a byte (unsigned).
0x2A	Cast to Word	Changes the top of the stack to a word (signed).
0x2B	Cast to Unsigned Word	Changes the top of the stack to a word (unsigned).
0x2C	Cast to Double Word	Changes the top of the stack to a double word (signed).
0x2D	Cast to Unsigned Double Word	Changes the top of the stack to a double word (unsigned).
0x2E	Cast to Float	Changes the top of the stack to a float (signed).

STRINGS

These instructions provide special functionality for strings. A string is a sequence of ASCII characters ending with a value 0x00. Because a string can consist of several bytes, only the address of the string can be placed on the stack.

Some operations will allocate system memory to hold a string. If these strings are no longer needed (for example when popped off the stack), this memory is freed. However, the memory is only freed if the Free flag is set. This is to differentiate between strings allocated by the script and constant strings.

Byte Code	Name	Operand	Description
0x2F	Push Constant String	Byte[] = String	Push the address of the first character on the stack. The Free flag is clear, because the memory is fixed and part of the script.
0x30	Push String Variable	Byte = Variable Index	Push the address contained in the string variable [Variable Index] on the stack. The Free flag is clear, because memory will be freed once the variable is discarded.
0x31	Pop String Variable	Byte = Variable Index	The address in the top entry of the stack is popped into the string variable [Variable Index]. The Free flag is also copied into the variable's flag.
0x32	String Equal	None	Compare the top two stack entries (must be strings) and replace them with true if they match exactly, false otherwise.
0x33	String Contains	None	Search for string(Top) in string(2nd From Top) and replace them with true if found, false otherwise.
0x34	String Concatenate	None	Replace the top two stack entries (must be strings) with a new combined string equal to [string(2nd From Top):string(Top)].
0x35	Value to String	None	Converts numeric value on stack to a string representation

MATH

These instructions perform mathematical operations on the top or top two stack entries. The result will maintain the type of the top stack entry. For example, 100.22(float) divided by 2(byte), will result in 50.11(float). However, 100(byte) divided by 2.6(float) will result in 38(byte), not 38.4615(float).

Byte Code	Name	Description
0x36	Increment	The top of the stack is incremented by 1.
0x37	Decrement	The top of the stack is decremented by 1.
0x38	Add	The top two stack entries are replaced with the result of ($Top + 2nd\ from\ Top$).
0x39	Subtract	The top two stack entries are replaced with the result of ($Top - 2nd\ from\ Top$).
0x3A	Multiply	The top two stack entries are replaced with the result of ($Top \times 2nd\ from\ Top$).
0x3B	Divide	The top two stack entries are replaced with the result of ($Top \div 2nd\ from\ Top$).
0x3C	Modulus	The top two stack entries are replaced with the result of (Top mod $2nd\ from\ Top$).

BITWISE OPERATIONS

These instructions perform bitwise operations on the top or top two stack entries. Note that these instructions are not intended for floating point values.

Byte Code	Name	Description
0x3D	Bitwise AND	Performs the bitwise AND operation on the top two stack entries and replaces them with the result.
0x3E	Bitwise OR	Performs the bitwise OR operation on the top two stack entries and replaces them with the result.
0x3F	Bitwise XOR	Performs the bitwise XOR operation on the top two stack entries and replaces them with the result.
0x40	Bitwise NOT	Performs the bitwise NOT operation on the top stack entry and replaces it with the result.

FLOATING POINT

These instructions perform special operations on floating point values.

Byte Code	Name	Description
0x41	Round	Performs mathematical rounding on the value. (For example: 12.4 → 12 and 12.5 → 13)
0x42	Floor	The value is rounded down to closest integer. (For example: 12.8 → 12)
0x43	Ceiling	The value is rounded up to closest integer. (For example: 12.2 → 13)

SPECIAL

The rest of the instructions do not fall into a specific class. They are listed below:

Byte Code	Name	Operand	Description
0x00	No Operation	None	Performs no action.
0x44	System Call	Byte = Call Index	Performs a system call to function [Call Index].

APPENDIX D: SYSTEM CALLS

A system call is the mechanism with which the virtual machine can access the underlying system functionality. For example, access to hardware or memory allocation.

Note that for every system call, there is also a firmware function that actually performs the action. The system call is only the mediator between this function and the virtual machine.

CONFIGURATION

These system calls are intended to be called once at the start of the script. They configure the virtual machine so that the rest of the script can execute as intended. At this point, only two functions are needed and they allocate memory for the stack and used variables.

Index	Name	Parameters	Description
0x00	Resize Stack	Word = Stack Size	Reallocates the stack from the default 100 entries to [Stack Size].
0x01	Allocate Variables	8 × Bytes	Allocates memory for each variable type according to the corresponding byte parameter. The order of the bytes is: byte, word, double word, unsigned byte, unsigned word, unsigned double word, float and string variables.

BUFFERED INPUT AND OUTPUT

"Buffer Inputs" and "Write Outputs" system calls are the key to this section. These functions read the physical input ports into an internal memory buffer, or changes output ports to reflect an internal buffer. Any other system calls to inputs will reference these buffered inputs, instead of the state at the physical input port. Similarly any system calls to change the state of an output, will only take effect after the "Write Outputs" system call.

Index	Name	Description
0x02	Buffer Inputs	Read the physical input ports into an internal memory buffer.
0x03	Write Outputs	Change the output ports to reflect an internal memory buffer.

The functions to manipulate these memory buffers (indirectly the I/O ports) are summarized in the next table:

Index	Name	Parameters	Description
0x04	Get Digital Input (Buffered)	Byte = Port Index	Pushes the state of the digital input port [Port Index], stored in the buffer, onto the stack.
0x05	Get Analog Input (Buffered)	Byte = Port Index	Pushes the value of the analog input port [Port Index], stored in the buffer, onto the stack.
0x06	Get Digital Output	Byte = Port Index	Pushes the state of the digital output port [Port Index] onto the stack. This is not buffered, because this state cannot change until "Write Outputs" is called.
0x07	Set Digital Output (Buffered)	Byte = Port Index	Sets the digital output port [Port Index] state in the buffer to HIGH.
0x08	Clear Digital Output (Buffered)	Byte = Port Index	Clears the digital output port [Port Index] state in the buffer to LOW.
0x09	Toggle Digital Output (Buffered)	Byte = Port Index	Toggles the digital output port [Port Index] state in the buffer.

TIME

These system calls return the current system time in different formats. This is required for the special event feature, allowing it to be evaluated only at specific times.

Index	Name	Description
0x0A	Get Time	Returns on the stack: double word = time in seconds since epoch (1 January 1970)
0x0B	Get Year	Returns on the stack: word = year ex. 2014
0x0C	Get Month	Returns on the stack: byte = month (1 - 12) ex. 1 = January
0x0D	Get Week	Returns on the stack: byte = week (1 - 52)
0x0E	Get Day of Month	Returns on the stack: byte = day (1 - 31)
0x0F	Get Day of Week	Returns on the stack: byte = day (1 - 7) ex. 1 = Monday
0x10	Get Time of Day	Returns on the stack: double word = time of day in seconds ex. (0 = 00:00:00) to (86399 = 23:59:59)
0x11	Get Time of Day (Hours Part)	Returns on the stack: byte = Hours part of time of day ex. (0) to (23)
0x12	Get Time of Day (Minutes Part)	Returns on the stack: byte = Minutes part of time of day ex. (0) to (59)
0x13	Get Time of Day (Seconds Part)	Returns on the stack: byte = Seconds part of time of day ex. (0) to (59)

BIBLIOGRAPHY

[1] B.S. Blanchard and W.J. Fabrycky, *Systems Engineering and Analysis*, 4th ed. New Jersey, US: Pearson Prentice Hall, 2006.

[2] I.S. MacKenzie and R.C.-W. Phan, *The 8051 Microcontroller*, 4th ed. New Jersey, US: Pearson Prentice Hall, 2007.

[3] S. Brown and J. Rose, "FPGA and CPLD architectures: a tutorial," *Design & Test of Computers, IEEE*, vol. 13, no. 2, pp. 42-57, summer 1996.

[4] T.L. Floyd, *Digital Fundamentals*, 8th ed. US: Prentice Hall, 2003.

[5] K.T. Erickson, "Programmable logic controllers," *Potentials, IEEE*, vol. 15, no. 1, pp. 14-17, Feb/Mar 1996.

[6] J.K. Ousterhout, "Scripting: Higher Level Programming for the 21st Century," *IEEE Computer*, vol. 31, pp. 23-30, March 1997.

[7] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating system concepts*, 7th ed. US: John Wiley & Sons, Inc., 2005.

[8] S.R. Schach, *Object-Oriented & Classical Software Engineering*, 7th ed. NY, US: McGraw-Hill, 2007.

[9] (2010) The FreeRTOS Website. [Online]. http://www.FreeRTOS.org

[10] (2010) The uCLinux Website. [Online]. http://www.uCLinux.org

[11] (2010) The Eclipse Website. [Online]. http://www.eclipse.org/

[12] (2010) The GNUARM Website. [Online]. http://www.gnuarm.com/

[13] (2010) The YAGARTO Website. [Online]. http://www.yagarto.de/

[14] Telit. (2009) GE863-PRO3 Datasheet. [Online]. http://www.telit.com

[15] Microsoft. (2009) Microsoft Development Website. [Online]. http://msdn.microsoft.com

[16] Telit. (2008, July) Development Environment User Guide: For C/C++ applications in GE863-PRO³ with Linux. [Online]. http://www.telit.com

[17] P.J. Deitel and H.M. Deitel, *C++ How to Program*, 6th ed. New Jersey, US: Pearson Prentice Hall, 208.

[18] J. Niestadt. (1999, May) Implementing A Scripting Engine. [Online]. http://www.flipcode.com/archives/articles.shtml

[19] G. Rosenblatt. (2002) Creating a Scripting System in C++. [Online]. http://www.gamedev.net/reference/

[20] Atmel. (2007, March) AT91SAM7X-EK Evaluation Board for AT91SAM7X and AT91SAM7XC User Guide. [Online]. hhtp://www.atmel.com

[21] J van der Merwe, "Linux as a real-time operating system for safety critical embedded applications - unmanned aerial vehicle case study," North-West University, 2009.