PORTING AND DEPLOYMENT OF THE OBERON SYSTEM

TO THE RASPBERRY PI

An Undergraduate Research Scholars Thesis

by

HAIPING XUE

Submitted to the Undergraduate Research Scholars program
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Research Advisor: Dr. Riccardo Bettati

May 2016

Major: Computer Engineering

TABLE OF CONTENTS

ABSTRACT

Porting and Deployment of the Oberon System to the Raspberry Pi

Haiping Xue
Department of Computer Science
Texas A&M University

Research Advisor: Dr. Ricardo Bettati
Department of Computer Science

With the advent of low-cost programmable embedded systems (e.g. Raspberry PI, VoCore, Arduino) the need emerges for software development tools that are efficient simple to use and inherently safe. In this project, we investigate the applicability of type-safe general-purpose language for use in limited resources embedded environments. The difficulties in porting the Oberon operating system to the Raspberry Pi lies in the fact that there is not a reliable Oberon compiler for the ARM architecture. To deal with this dilemma, we decide to port the Oberon compiler to a commonly available platform where allow us to test and verify the correctness of the generated ARM binary code. Programming language C++ and x86 architecture becomes our choice. During the research, we invented a solution to cope with the difficulties due to the lack of a reliable Oberon compiler for ARM architecture.

CHAPTER I

INTRODUCTION

With the advent of low-cost programmable embedded systems (e.g. Raspberry PI, VoCore, Arduino) the need emerges for software development tools that are efficient simple to use and inherently safe. In fact the success of the Arduino Platform, for example, relies largely on its intuitive and therefore simple to use, programming environment. An Arduino program is represented as a sketch, which contains a setup function, which in turn initializes the state of the program and a loop function, which runs continuously afterwards. This programming paradigm has been very successful at enabling untrained users to use embedded processing in a variety of settings. This simplicity of use and inherent safety, comes at the cost of significantly limiting the expressiveness and capabilities of the language, such as lack of support for multiple processors, event driven processing and multithreading. This is particularly limiting embedded applications.

In this project, we investigate the applicability of type-safe general-purpose language for use in limited resources embedded environments. The hypothesis that we set out to prove is three-fold: First we will illustrate that the porting of a programming environment to a resource-constraint embedded system is possible with limited effort. Second, we will show that a fully type-safe general –purpose language can run efficiently on such a system. Third, we will illustrate how such an environment can be used to develop predictably safe software. Specially, we will use the Oberon [2] system as the programming environment and the Raspberry Pi [5] as the embedded platform. The Oberon system was originally developed in the early nineties to develop tightly interactive system on workstations. Due to its inherent simplicity, this system has recently

regained popularity for programming ultra-small embedded systems. Such as CPUs on small field-programmable arrays (FPGAs) [2]. An example of the latter is the Tiny Registers Machine (TRM) [1], which is a processor architecture specifically designed for use on FPGA's. It has been shown that the Oberon system can be run efficiently and with high performance on a system as small as the TRM. Moreover, Oberon can be easily extended to multiprocessor systems as well, thus paving the way towards high-performance limited-resources embedded systems.

CHAPTER II

BACKGROUND

Raspberry Pi

The Raspberry pi is a low cost, credit-card sized computer developed in 2012 at the University of Cambridge's Computer laboratory. The primary purpose of this computer is to help children learn programming in both the developed and the developing world [6]. This powerful mini-computer is built up with an ARM processor, 3D VideoCore IV graphics processor, 512MB RAM and a SD memory card connector. The HDMI socket, USB connectors for mouse and keyboard and an Ethernet connector make it possible to become a desktop PC. In addition to these basic features, the GPIO (general purpose I/O) connects give the Pi capability to communicate with sensors, switches, LEDS, and motors in embedded applications. With a Linux system that is compatible with the Pi, it is able to run as a web server, or a device controller [7]. Nowadays, more applications and projects are created upon on this little chip and get big ideas. The German firm All For Accounting has popped the Pi into a red-and-black box, and plan to

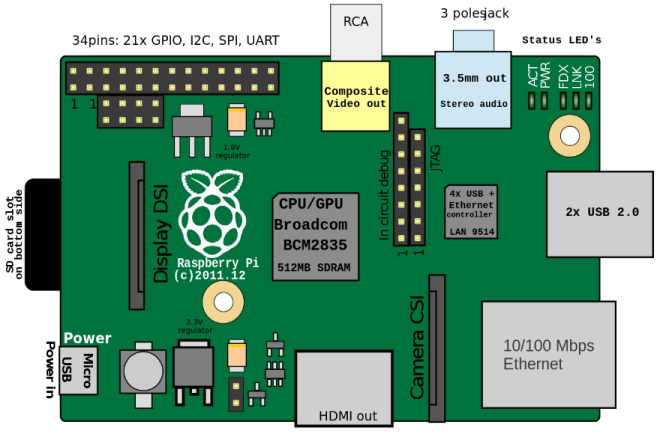

Figure1. Raspberry Pi [9]

sell it as a cheap computer for companies to run their stock-planning and accountancy software. The open API TargetR platform uses Raspberry Pi as a hardware player for Target digital signage. Another company, Ciseco, is currently utilizing Pi as the computing brain of a home-automation system. The Pi connects many devices in the home through low-power radio communication [8].

Project Oberon

In Project Oberon, a simple but complete desktop computer system is designed from scratch including an operating system, a compiler and a computer. The simplicity and clarity offers developers a chance to know and implement the entire system has enough power to be useful and usable by themselves. The structure of the operating system is hierarchical, and no cycles. As

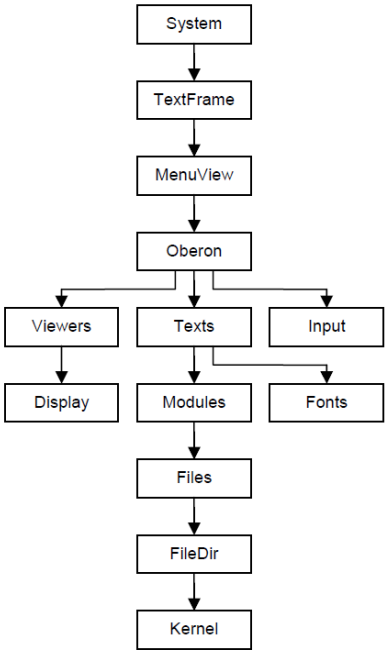

Figure 2. The Hierarchy of Oberon Operating System [10]

shown in figure 2, the modules on the top of the hierarchy allows user to communicate with tools and commands. The lowest modules of the hierarchy in the operating system, in contrast, are used to control the hardware underneath the operating system. Module Oberon is the heart of the system which contains the central loop for controlling returns after each command interpretation [10].

The RISC (Reduced Instruction Set Computer) processor designed for Project Oberon presents an architecture with limited but complete features. As shown in Figure 3, the main components of this RISC architecture include an ALU (arithmetic and logic unit), a control unit (instruction register and program counter), memory, a register bank and four flag registers N, Z, C and V. The core unit, ALU, features a bank of 16 registers with 32 bit words each. The arithmetic and logical operations decoded from assembly always operate on these registers. Each instruction takes a single clock cycle to execute except memory, multiplication and division instructions. Data transfer between memory and register is done by load and store instructions [10].

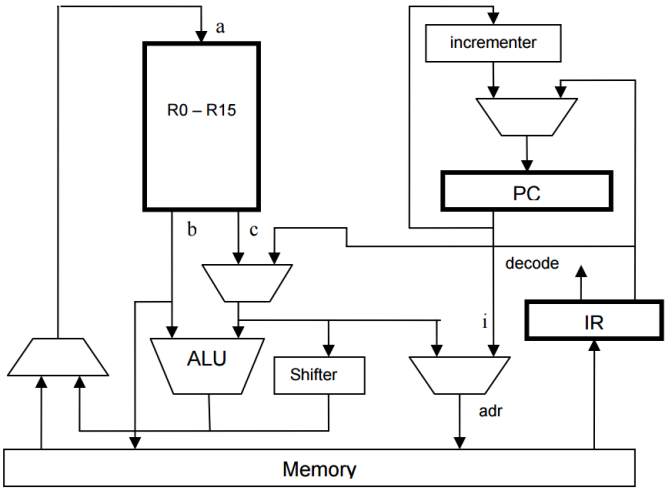

Figure 3. RISC Architecture [10]

ARM Architecture

ARM (Advanced RISC Machine), is a reduced instruction set computer architecture invented by British company ARM Holdings. ARM cores are used in many portable products, including Apple's iPads, iPhone, Canon PowerShot digital cameras, iPod, Nintendo DS, BeagleBone single board computer and other. The ARM architecture incorporates many typical RISC architecture features: a large uniform register file, a load/store architecture, simple addressing modes and uniform and fixed-length instruction fields. In additional these basic features, the ARM provides extra optimized operations to maximize data and execution throughput. There are 16 visible 32-bit registers and three of them have special purposes: Register 13 are used as a stack pointer, Register 14 is the Link Register and Register 15 is the Program Counter. The remaining 13 registers are all general purpose registers. All processor states are held in status registers [11].

CHAPTER III

AN OBERON COMPILER FOR RASPBERRY PI

Overview of the Existing Compiler

To understand porting strategy in this paper, compilation mechanism of the C style Oberon compiler need to be briefly illustrated. The C style Oberon complier is descent recursive parser built on Oberon Complier written in Language Oberon. The compiler is built into three main parts: parser, generator and binary code. The parser will first parse through code written in language Oberon, extract the parameters in the source code, store the parameters in the symbol table and pass the information to the generator. The generator will then analyze the parameters and information retrieved from the parser and decide a set of binary code need to be generated for a particular line of code in source code [3].

Figure 4. Original Oberon Compiler

In this project, the translation of instruction sets from Oberon RISC to ARM is done by adding

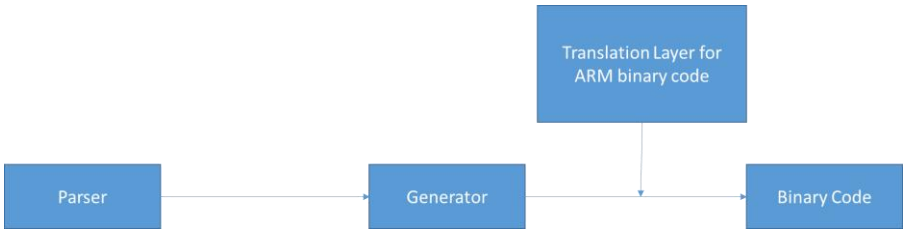

Figure 5. New Compiler with extra layer

an extra layer between the generator layer and binary code layer, which is shown in Figure 2.

Oberon RISC Instruction Set

In the Oberon RISC architecture instruction set, there are four types of instructions and instruction formats. They are register instructions, immediate instructions, memory instructions and branch instructions respectively. The instructions are 32-bit long each. The register instructions only utilize registers in operations. The result are stored in destination registers after execution. The memory instructions fetch and store data in memory. Branch instructions update the address in program counter [10].

ARM Instruction Set

The ARM instruction set has six operating modes: user, FIQ, IRQ, supervisor, abort, undef. Although there are many instruction formats in ARM instruction set, only four types of instructions and instruction formats are used in this research. They are data processing instructions, multiply instructions, load/store byte/word instructions and Branch instructions. The data processing instruction contains arithmetic operations, comparisons, logical operations and data movement between registers (MOV instruction). The second operand of the instruction can be applied to a barrel shifters in order to finish shift operations. Multiply instruction simply allows an arithmetical multiply operation by using registers. The memory instructions fetch and store data in memory with the option of using a memory address in a register or a PC-relative address in immediate field. Branch instructions change the address in program counter [11].

Comparison

The ARM and Oberon RISC's instruction set share some common points. The functionality of data processing instructions in ARM are similar to Oberon RISC's register operations except the fact that there is no real shift operations in ARM. The branch instructions' offsets in both instruction sets have the same jump range. However, there is also a number of variances between two instruction set that makes the porting process more difficult. The first issue is the 8-bit immediate value in ARM data processing instructions. The ARM architecture only allocates 8 bits for the immediate value in data processing instructions while Oberon RISC architecture has 16 bits for its immediate value. Although the ARM architecture can use barrel shifter to solve part of this problem, certain values can not be achieve with a single shift operation. In the memory operations, offset size of two architectures also varies. The second issue is that there are some operations in Oberon RISC instruction set that do not exist in ARM instruction set. Therefore, multiple instructions have to be used in order to finish a single instruction in Oberon RISC instruction set. The third issue is certain ARM instructions do not have an immediate format. Extra work has to be done for moving the immediate value into a proper register so as to complete a single instruction in Oberon RISC instruction set.

From Oberon RISC to ARM

Figure 3 illustrated the method that is used to translate instruction set in Project Oberon to ARM instruction set and to generate binary code for the ARM architecture. In order to minimize the numbers of changes in the code, a number of transfer functions which are named as aput() have been added inside all the put() functions. In these functions, different types of binary codes in Project Oberon's RISC processor are translated into ARM binary codes. The op code and

registers are assigned following the ARM's binary code format. In addition, control bits need to be assigned with proper values to prevent unexpected behaviors. However, some instructions in Project Oberon may not be able to match a particular ARM binary instruction. This will be handled separately by converting binary instructions into one or multiple ARM binary instructions to perform the same functionality for the original binary codes. For instance, there is not an ANN (and not) instruction in the ARM instruction set. To translate this instruction, we used a MVN (move not) and a AND instruction in ARM instruction. The next step is to check all the offsets in ARM architecture and find out a way to deal with short offsets problems in ARM binary instructions.

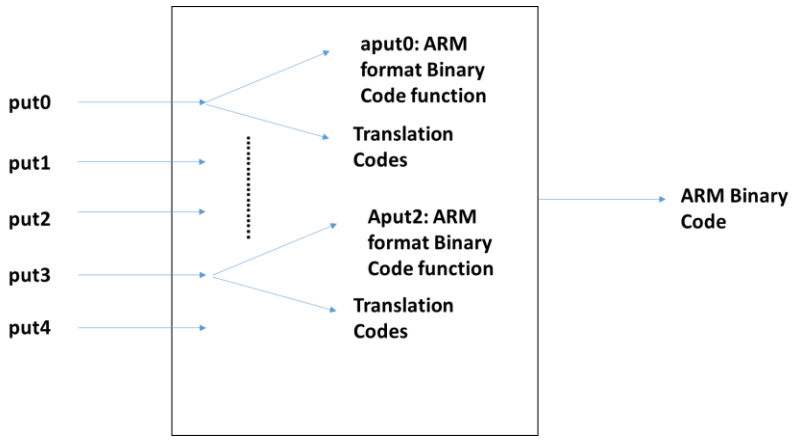

Figure 6. Method of Translation

CHAPTER IV

PRACTICABILITY OF PORTING THE OBERON COMPILER

The Oberon Compiler

Computer programs are text created by programmers following certain grammar and syntax for a certain programming language. However, the machine is not able to understand this high-level programming language. Appropriate machine instructions, binary code, need to be generated before the human-written text can be processed by a computer. The program that is used to generate these machine instructions is called a compiler. The Oberon compiler in Project Oberon is a recursive descent parser that is used for this purpose. The translation process is divided into four parts: lexical analysis, syntax analysis, type check and code generation which is shown in Figure 7. The computer program will firstly be translated into a set of symbols for certain language in lexical analysis stage. Then the syntax analysis will build the symbols into a presentation according to the syntactic structure of the computer program. Then, the syntactic

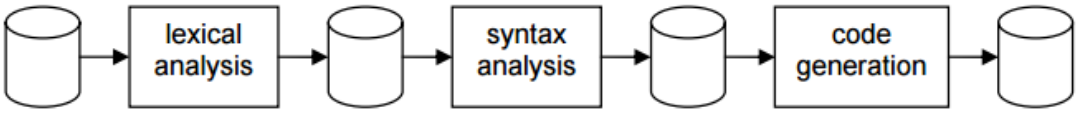

Figure 7. Code generation process [3]

rules and compatibility rules of the language will be verified in syntax analysis. Finally, the code generation stage will generate binary codes for the destination computer. In the Oberon compiler, a front end and a back end are specified in order to separate the parsing and code generation. The front end is aim at parsing the source code and building a syntactic structure, syntax tree, based

on its source code. The back end is responsible for code generation for specific computer. Therefore, the front end will not be changed when programmer switch the target computer [3].

Difficulties and Solution

The difficulties in porting the Oberon operating system to the Raspberry Pi lies in the fact that there is not a reliable Oberon compiler for the ARM architecture. To deal with this dilemma, we decide to port the Oberon compiler to a commonly available platform where allow us to test and verify the correctness of the generated ARM binary code. Programming language C++ and x86 architecture becomes our choice. The next step is to port the Oberon compiler on x86 host to ARM architecture. This step allow us to generate ARM binary code with the Oberon compiler on an x86 host. With the new compiler that generate ARM binary code, we can compile the Oberon Operating System into ARM binary code and finish porting the Oberon Operating System to the Raspberry Pi. The Last step is to substitute the RISC Oberon compiler in the Oberon operating system to ARM Oberon compiler. In this way, we will obtain a self-contained Oberon operating system that can run and compile programs on the Raspberry Pi. Figure 8 illustrates our approach with a diagram.

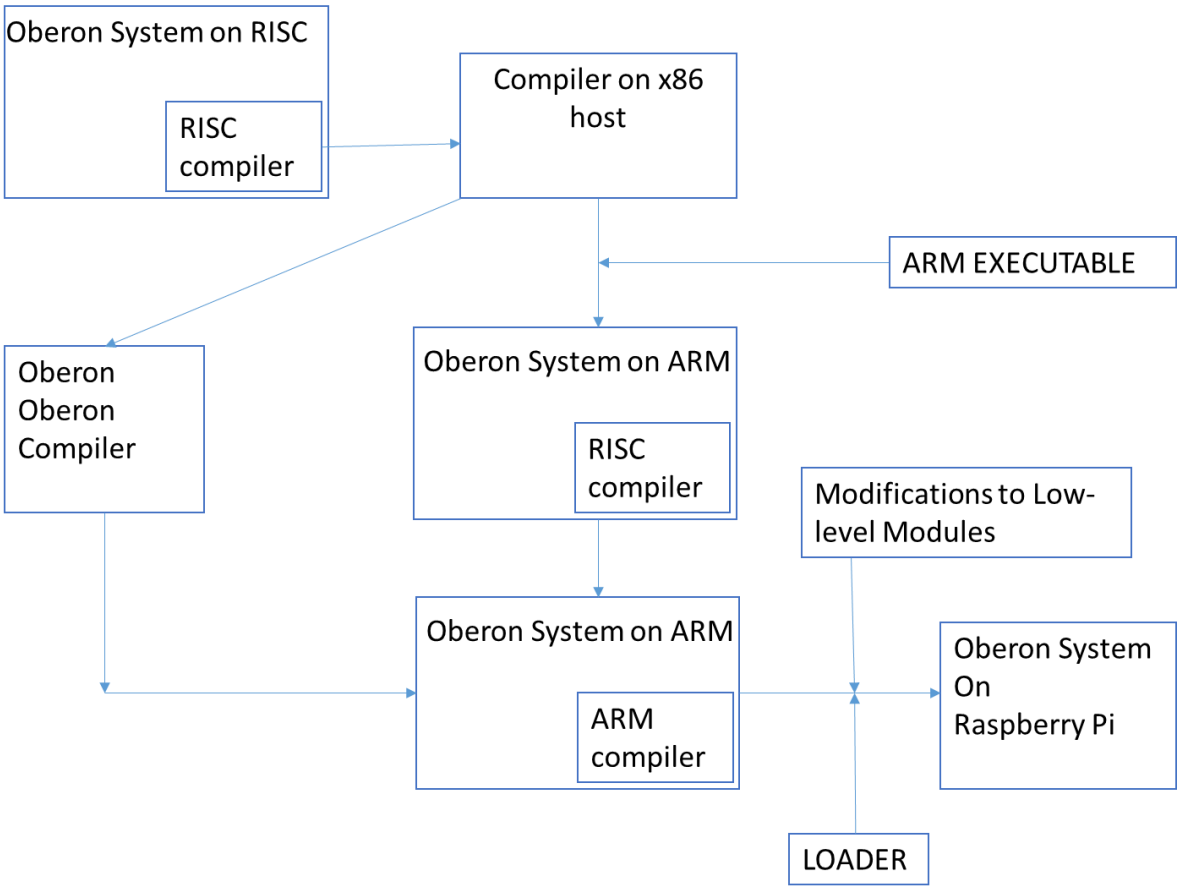

Figure 8. Approach to port the Oberon System to the Raspberry Pi

CHAPTER V

CURRENT STATUS AND REMAINING WORK

Current Status

At the current stage, a C style Oberon compiler for the ARM architecture is built on the x86 host. This compiler is able to generate a reasonable set of ARM binary code. We compiled some simple programs using the C style Oberon compiler to test the logical correctness of the generated binary code. Figure 9 below shows us a sample result of a small program. In this program, the procedure assigns number 11 to the integer type variable i and returns it. Starting

```
MODULE Example;

   PROCEDURE ArraySum*(r: INTEGER):
INTEGER;

   VAR i: INTEGER;

   BEGIN

    i:=11;

    RETURN i

   END ArraySum;
```

```
Decode Object File: example.rsc

Example  3484B1A3   1    72
imports:
type descriptors

data     0
strings

code
   0     024DD00C       SUB SP  SP         12
   1     058DE000       STR   LNK  SP        0
   2     058D0004       STR    R 0 SP        4
   3     03A0000B       MOV  R 0  R 0        11
   4     058D0008       STR    R 0 SP        8
   5     059D0008       LDR   R 0 SP         8
   6     059DE000       LDR  LNK  SP         0
   7     028DD00C       ADD SP  SP         12
   8     01A0F00E       MOV PC   R 0 LNK
   9     024DD004       SUB SP  SP          4
  10     058DE000       STR   LNK  SP        0
  11     059DE000       LDR  LNK  SP         0
  12     028DD004       ADD SP  SP          4
  13     01A0F00E       MOV PC   R 0 LNK
commands:
entries
    36     0
pointer refs

fixP =         0
fixD =         0
fixT =         0
entry =        36
SMB done (type char to end)
```

Figure 9. Example source code and binary code

from the top, we can see that the binary code firstly moves down its stack pointer and stores the link register value and old value of R0 on the stack. Then it moves the new value 11 into R0 and stores new value of R0 on the stack as the return value. This means that R0 is used to hold the value of integer type variable i in this procedure. After finish all the operations in the procedure,

the program load the old value of link register and R0 from the stack and increment the stack pointer to finish the return process.

Remaining Work

There are still some remaining work need to be done before running the Oberon operating system on the Raspberry Pi. Since the Oberon RISC architecture and ARM architecture is not the same, some low-level modules need to be. For example, ARM architecture has a program control register to control privilege and user modified to accommodate ARM architecture mode. In Oberon RISC architecture, there is not such a register. Therefore, we need to add control logics to low-level module to handle the program control register. Another important issue is to find a suitable loader to load the operating system onto the Raspberry Pi. All these work need to be done before we can run the Oberon operating system on the Raspberry Pi.

CHAPTER VI

CONCLUSION

In this research, we investigate the applicability of type-safe general-purpose language for use in limited resources embedded environments. The embedded system environment we choose is the Raspberry Pi platform and the language we choose is Language Oberon. During the research, we invented a solution to cope with the difficulties due to the lack of a reliable Oberon compiler for ARM architecture. We ported the Oberon compiler to the x86 host and the ARM architecture. The test result of the C style compiler is logically correct for some simple programs. More complicated programs need to be test out to improve the C style Oberon compiler. The future work for us is to modify low-level modules in the Oberon operating system to accommodate the ARM architecture and find a loader to load the Oberon operating system onto the Raspberry Pi.

REFERENCES

[1] Niklaus Wirth (2010 Aug). Experiments in Computer System Design. Retrieved from http://www.inf.ethz.ch/personal/wirth/FPGA-relatedWork/ComputerSystemDesign.pdf

[2] Gutknecht, Jürg, and Niklaus Wirth. "Project Oberon-The Design of an Operating System and Compiler." (1992).

[3] Niklaus Wirth (2014 Feb). Compiler Construction. Retrieved from https://www.inf.ethz.ch/personal/wirth/CompilerConstruction/CompilerConstruction1.pdf

[4] Seal, David. ARM architecture reference manual. Pearson Education, 2001.

[5] Upton, Eben, and Gareth Halfacree. Raspberry Pi user guide. John Wiley & Sons, 2014.

[6] Bush, Steve. "Dongle computer lets kids discover programming on a TV."Electronics Weekly. http://www. electronicsweekly. com/Articles/2011/05/25/51129/Dongle-computer-lets-kids-discover-programming-on-a. htm. Retrieved 11 (2011).

[7] Brock, J. Dean, Rebecca F. Bruce, and Marietta E. Cameron. "Changing the world with a Raspberry Pi." Journal of Computing Sciences in Colleges 29.2 (2013): 151-153.

[8] Edwards, Chris. "Not-so-humble raspberry pi gets big ideas." Engineering & Technology 8.3 (2013): 30-33.

[9] Raspberry Pi B+ Rev. Digital image. Wikipedia. N.p., 5 Apr. 2015. Web.

[10] Wirt, N., and J. Gutknecht. "„Project Oberon “." (1992).

[11] Seal, David. ARM architecture reference manual. Pearson Education, 2001.