University of Koblenz – Landau

Faculty 4

Institute of Computer Science

Apple][Emulation on an AVR Microcontroller

August 2014

Bachelor thesis

to obtain the academic degree

“Bachelor of Science (BSc)”

Submitted by: **Maximilian Strauch**

(Student ID: 211 201 869)

Supervised by: **Prof. Dr. Hannes Frey**

Dr. Merten Joost

Fiat lux!

Ehrenwörtliche Erklärung

Hiermit bestätige ich, dass die vorliegende Arbeit von mir selbständig verfasst wurde und ich keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet–Quellen – benutzt habe und die Arbeit von mir vorher nicht in einem anderen Prüfungsverfahren eingereicht wurde. Die eingereichte schriftliche Fassung entspricht der auf dem elektronischen Speichermedium (CD-Rom).

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	✗	
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	✗	

....

(Ort, Datum) (Unterschrift)

Apple][Emulation on an AVR Microcontroller

***Abstract* –** The Apple][computer was one of the first three completely assembled systems on the market. It was sold several million times from april 1977 to 1993. This 8 bit home computer was developed by Steve Wozniak and Steve Jobs. They paved the way for the Apple Macintosh computer and the nowadays well known brand Apple with its products.

This thesis describes the implementation of a software emulator for the complete Apple][computer system on a single Atmel AVR microcontroller unit (MCU). The greatest challenge consists of the fact that the MCU has only a slightly higher clock speed as the Apple][. This requires an efficient emulation of the CPU and the memory management, which will be covered later on along with the runtime environment controlling the emulator. Furthermore the hardware implementation into a handheld prototype will be shown.

In summary this thesis presents a successful development of a portable Apple][emulator covering all aspects from software design over hardware design ending up in a prototype.

Keywords: Apple][, emulation, Atmel AVR microcontroller

Emulation eines Apple][auf einem AVR Microcontroller

***Zusammenfassung* –** Der Apple][war einer der drei ersten kompletten Computersysteme auf dem Markt. Von April 1977 an wurde er rund 16 Jahre lang mehrere Millionen mal verkauft. Entwickelt wurde dieser 8 Bit Homecomputer von Steve Wozniak und Steve Jobs. Sie ebneten damit den Weg für den Macintosh und das heute gut bekannte Unternehmen Apple.

Diese Arbeit beschreibt die Implementierung eines Softwareemulators für das komplette Apple][Computersystem auf nur einem Atmel AVR Microcontroller. Die größte Herausforderung besteht darin, dass der Microcontroller nur eine geringfügig höhere Taktrate als die zu emulierende Hardware hat. Dies erfordert eine effiziente Emulation der CPU und Speicherverwaltung, die nachfolgend zusammen mit der Laufzeitumgebung für die Emulation vorgestellt wird. Weiterhin wird die Umsetzung des Emulators mit Display und Tastatur in Hardware näher erläutert.

Mit dieser Arbeit wird die erfolgreiche Entwicklung eines portablen Apple][Emulators, von der Software über die Hardware bis hin zu einem Prorotypen, vorgestellt.

Schlagworte: Apple][, Emulator, Atmel AVR Microcontroller

Table of contents

Chapter 1: Preface 5

1.1 Related work 8
1.2 Structure 9
1.3 Legal thoughts 9
1.4 Terminology 10

Chapter 2: Essentials 11

2.1 The Apple][........ 11
2.1.1 MOS Technology 6502 12
2.1.2 Memory organisation 19
2.1.3 Video output 22
2.1.4 The keyboard 24
2.1.5 Other hardware features 25
2.1.6 Software insights: System Monitor & BASIC 25
2.2 Microcontrollers vs. Microprocessors 26
2.3 Different architectures 27
2.3.1 Von Neumann architecture 27
2.3.2 Harvard architecture 27

Chapter 3: Software implementation 29

3.1 Concept and basic setup 29
3.2 Emulation of the MOS 6502 CPU 30
3.2.1 Requirements and exclusions 30
3.2.2 Designing the emulator – a simple approach 32
3.2.3 Revision of the first approach 35
3.2.4 Internals of the instruction opcode implementations 36
3.2.5 Identifying other retardants 39
3.2.6 The memory access 40
3.2.7 First tests 43
3.2.8 Going back to the roots 44
3.2.9 New speed measurements & summary 52
3.3 The emulator runtime environment 53
3.3.1 The structure 54
3.3.2 Display output of the emulator (module "Display") 55
3.3.3 Keyboard input (module "Keyboard") 56
3.3.4 Bringing software into the emulator (module "DSK I/O") 58
3.3.5 Hibernation feature (module "State I/O") 61
3.3.6 Sound output 62
3.3.7 Emulator backend UI insights 62

Chapter 4: Hardware implementation ... 64

4.1 The “emulation” microcontroller ... 64

4.1.1 Pinout and pin mapping ... 65

4.1.2 Interfacing the display ... 66

4.1.3 Talking to the EEPROM ... 69

4.1.4 The SD card and ISP connectors ... 70

4.2 The “keyboard” microcontroller ... 72

4.2.1 Pinout ... 72

4.2.2 Keyboard switch matrix design ... 73

4.2.3 Software UART transmit ... 74

4.2.4 Possible disadvantages ... 75

4.3 BOM ... 75

4.4 The prototype ... 77

4.5 Schematic of the prototype ... 78

Chapter 5: Conclusion & Outlook ... 80

5.1 Conclusion ... 80

5.1.1 Achieved emulator speed ... 82

5.1.2 Unmentioned aspects ... 83

5.2 Further development ... 84

Chapter 6: Appendix ... 86

6.1 Glossary ... 86

6.2 Bibliography ... 89

6.3 “Speed” measurement setup ... 91

Chapter 1: Preface

Introduction

The market of microcontrollers grows from year to year and expanded in the last ten years by 80% [1]. Those tiny devices, equipped with everything a modern computer contains, manage many things in our daily life. Because of the fact that they are based on a ➝RISC instruction set architecture, ➝microcontrollers are so powerful that the question arises whether it can emulate an entire historical personal computer.

A very interesting personal computer is the Apple][, which was build by Steve Wozniak together with Steve Jobs and sold from the beginning of the late 70th. It was a very popular and powerful system, providing very sophisticated graphical and technical features for this time. Together with two other computer systems, it was the first complete assembled computer system commercially available. Other computer systems were only available as self-assembly kits and not so powerful. It was build upon an 8 bit architecture, using the – for those days – famous MOS Technologies 6502 ➝microprocessor (CPU), which was widely used in many computer related devices in the late 70th and 80th. Famous devices that make use of the 6502 are: the Apple][computer series, the Atari 2600 game console, Nintendo NES game console (slightly modified 6502 model), the Commodore VIC-20 and many other devices [2].

Since the aimed device does not only emulate the 6502 microprocessor, but also perform display output and other I/O to achieve an emulation of the entire computer system by a microcontroller, it is too complex to evaluate it, using a theoretical model. The research question is therefore not only to find out, if it is possible, but also to find out the limitations which arise.

Motivation

This project provides the feasibility to investigate the facets of hardware related computer science. This part plays an important role in the everyday life of everybody. Tiny, embedded devices like microcontrollers are embedded in nearly every electronic device. From the washing machine over the electronic radiator thermostat to a car. Inside the play field of microcontroller devices one can learn and understand easily, how modern computers work and see, by example, the basic concepts of computers. This “experiments” can be done very easily on microcontrollers, because they are very simple in structure and so it is easy to get started in comparison to the x86 system architecture. In addition to this, one will discover different aspects of communication between electronic devices and learn to understand and use protocols like: ➝UART,

→SPI or →TWI. Those protocols are not limited to the world of microcontrollers: computers rely on this protocols by example to monitor the processor temperature using sensors, which stream the data through the TWI protocol to the CPU. Or more famous: the SD card can be accessed through the SPI protocol. This leads to the fact that the world of microcontrollers is important for modern hardware and devices. By getting an insight into this “world”, one is capable of understanding computers and their structure even better.

Developing a handheld device from scratch, which can be hold in one's own hands and driven without the need of a computer, makes software real and touchable. Combining this with a recreation of a historical important device, like the Apple][, allows to preserve nostalgia and brings the history of computer science to the present.

By learning many things about computers, microcontrollers and their structure and creating a handheld device from scratch, to take a look into the early days of computers and their usage, this is an ideal project for the purpose of an bachelor thesis.

What is emulation? And what's new with that?

Emulation is the process of a very precise simulation of original hardware, in order to use software which was compiled for the original hardware, through the emulation on different hardware which is incompatible [3].

In this case, the software written for the Apple][computer only works on exactly this processor which was used by the Apple][. By creating an emulator of the Apple][one needs to simulate the processor in order to let the software, written for the Apple][, work on the emulator which runs on a completely different hardware.

But there is nothing new: by browsing through the world wide web one can find more than enough emulators for all kinds of historical devices and also Apple][emulators. These emulators work on a computer with a lot of computational power and they are not very portable.

The idea of this thesis is to create an emulator in software and hardware, especially designed only for the emulation of the Apple][system. The created device should be a “handheld” device which is portable. The used microcontroller to run the emulator relies – just like the Apple][– on an 8 bit architecture and has only a slightly faster clock of 20 MHz versus 1 MHz of the Apple][.

There is no doubt that the emulator works with a normal computer, decreeing over multiple cores and many giga hertz of CPU speed. An important question is if the emulation will work on a tiny microcontroller, which is only twenty times faster than the emulated host, the Apple][. And the device will not be finished with the emulation of the Apple][microprocessor. Display output, keyboard input and some other features need to be implemented to make the device useable.

Talking about emulation is talking about speed. By recreating the hardware of the Apple][with a software emulator, control overhead data is generated, which must be processed and stored. The more sophisticated the architecture is, the more overhead is generated and the calculation time grows.

Besides the software part, this thesis features also the hardware implementation pro-

cess of creating a physical device, using skills like soldering and building, facing different problems which will arise from the electronic part of the project.

FPGAs as competitors?

A ➝FPGA can be used to less emulate but more ***"be"*** the target CPU of any device, by example the Apple][computer. It can also imitate an entire system consisting of multiple microchips. And it was already used by different projects aiming an emulation of the Apple][, which can be found in the world wide web [4].

The "field programmable gate array" (FPGA) is a "programmable" integrated circuit. Using a computer program a logical circuit, out of discrete logical gates, is created and then programmed or build to the FPGA chip. After programming the FPGA actually ***is*** the circuit, designed on the computer and realizes it in hardware. This allows various projects and also the perfect imitation – not emulation, because it is rebuilt out of discrete logical components – of CPUs [5] (p. 21).

As seen by this brief introduction, FPGAs are a lot more powerful than microcontrollers and can be used for sophisticated tasks like realtime image processing, by example. Beside the fact that they are more cost intensive, the aim of this thesis is to use a microcontroller to get an idea of how far the computational power can be lowered to emulate the system. But it is also a challenge of costs, aiming to construct a device which uses less components and is as cheap as possible.

Other embedded systems

Modern devices, like cell phones or tablets are using mostly ➝ARM processors. Those are also available as chips to develop embedded applications. But they are baed on a 32 bit architecture and there is – just like the FPGAs – no motivation to try this project with such a sophisticated architecture, because there is no challenge to develop the emulation software and face problems. One could simply bring a Linux system onto this chip and run emulator software from the web.

Demand profile

Due to the fact that this project is a thesis with the aim of an Apple][emulator, some requirements were defined at the beginning of the project to ensure a high quality result.

First of all, the overall target of this thesis is to build an emulator handheld device of the original Apple][model from 1977 with a memory configuration of 12KB[1]. The other key requirements are:

- **implementation of a 6502 microprocessor *without* the decimal mode in C or assembly language**
- **interfacing a TFT display with video RAM**
- **sketching a custom keyboard with controller**
- **realization the Von Neumann architecture on the Harvard architecture of the microcontroller**

1 This restriction will be discussed in section 3.1 "Concept and basic setup", subsection "Hardware limitations" (p. 30).

- **implementation of different memory accesses (RAM, ROM, I/O)**
- **software loading possibility** (for programs written for the Apple][)
- **buildup as a mobile handheld system**
- **documentation of the result**

Those requirements are evaluated in detail on the end of this work with the gathered results in chapter 5 “Conclusion & Outlook“ (p. 80).

1.1 Related work

After an advanced research, only two other projects could be found, which rebuild the Apple][computer system – or parts of it – on an microcontroller, especially an AVR microcontroller (which will be used later on). Both were implemented at the Cornell University (Ithaca, NY, United States of America) during the course “ECE 4760: Designing with Microcontrollers” [6].

In the year 2007, three students from Cornell University tried to develop a system, which is able to emulate the Apple][computer on an Atmel AVR ATMega32 microcontroller, in a practicum [7]. Because of the fact that they had not enough time to finish their project and had some issues with the memory subsystem, only a working MOS 6502 microprocessor emulator, a memory subsystem and a partial GPU (graphic processing unit) was created by them without reaching the desired Apple][emulator system. The difference to this thesis is not only the fact that this thesis creates a running device, but also that this thesis creates a fully self-contained Apple][emulator with less hardware components and equipped with a display to be portable.

Later on, in the year 2009, another team of two students tried to create an emulation of the NES (Nintendo Entertainment System) [8]. Despite the fact that this project was not led to the target of a complete NES emulation, which is hardly at the upper end of the possibilities of the Atmel AVR microcontroller due to the complex PPU, this project shares only the MOS 6502 processor emulation with this project.

Other MOS 6502 microprocessor emulations, based on an AVR microcontroller, are available on the world wide web [9] (only *one* example page). The most MOS 6502 microprocessor emulations found on the web are written in C or C with inline assembler and not highly optimized which will turn out as a key factor later on. Because of this fact, using an existing 6502 microprocessor emulator is not an option.

During further research, it turned out that – to the knowledge of the author – nobody had ever tried the target of this project: to create a portable Apple][emulator handheld device. The shown other projects did not reach their target and result in an unusable emulator device. So the results might only be used for initial design but not for real implementation purposes.

Furthermore there is no indicator that anybody has created an Apple][computer system emulator combining a custom keyboard, display, batteries and the microcontroller in one single device that can be carried around in a pocket. And this is exactly the desired result of this thesis. Due to this fact, the proposed solution of this thesis is a premiere.

1.2 Structure

This thesis features the entire way of producing a final product, which would be done by companies: from the idea over first thoughts to the software development and finally the implementation of a working prototype. The document is structured as follows.

First of all the structure and details of the Apple][computer system and the MOS Technology 6502 microprocessor are explained in chapter 2 (p. 11). Due to the fact that one needs to know all the details of the microprocessor or Apple][system to emulate it, those are deeply covered. Then the implementation of the emulator software is described in chapter 3 (p. 29). Thereby some first considerations for the hardware of the resulting device are made, since the software implementation of the emulator is limited by hardware details of the used microcontroller. This chapter not only describes the implementation of the 6502 microprocessor emulation, it also describes other parts of the emulator like rendering an Apple][screen to the attached display and the idea of an emulator runtime environment which manages the emulator. In chapter 4 (p. 64) the software implementation is supplemented by hardware implementation details, forming the aimed portable handheld Apple][emulator (chapter 4.4 "The prototype", p. 77). Finally, this thesis is closed with a conclusion of the achieved targets: it takes an outlook to new features which might be implemented to advance the resulting handheld device in chapter 5 (p. 80).

1.3 Legal thoughts

Emulating a system opens the question of legitimacy. In this particular case, there is no reason for concern. To the knowledge of the author, the MOS 6502 was not patented itself but some features were, like the on-the-fly correction of binary adding results which was registered on the 16th september 1975 [10] (US patent: US 3991307 A). Since this patent lasts up to 20 years and MOS Technologies no longer exists, there is most likely no active copyright left [11]. One should also not overlook that this work has exclusively an educational purpose.

Furthermore the software and operating system of the Apple][is the critical component, since this parts are copyrighted by Apple Inc. and others. A usage of this software might not be allowed. But since a marvellous "donation" from Apple Inc. to the Computer History Museum in Mountain View (CA, United States of America) the source code was made available for non-commercial use at the end of 2013:

> *With thanks to Paul Laughton, in collaboration with Dr. Bruce Damer, founder and curator of the DigiBarn Computer Museum, and with the permission of Apple Inc., we are pleased to make available the 1978 source code of Apple II DOS for non-commercial use. This material is Copyright © 1978 Apple Inc., and may not be reproduced without permission from Apple* [12].

The downloads contain various documents related to the Apple][development and the source code for the Apple][DOS and BASIC.

(Thanks, Apple!)

1.4 Terminology

To prevent misunderstandings and irritations some terms, symbols and spellings are expounded in the following:

- if furthermore memory sizes are meant, “K” is an abbreviation for “kilobyte”. So the term “8K” stands for 8 kilobyte of data (8.192 byte).
- strings starting with “`0x`” indicate a hexadecimal number. Strings starting with “`0b`” indicate a binary number. An asterisk “*” in a binary or hexadecimal number indicates a placeholder for any value of this number system, e.g. `0xf*` stands for numbers from `0xf0` to `0xff`. If there is no such notation, the numbers are noted in the standard decimal system to base ten.
- humans tend to start counting by one. On everything related to technical topics, numbering starts by zero. This is the way, how numbering is handled in this document. Be aware of the fact that ordinal numbers still start by one.
- all abbreviations or words with a “➝” character in front have a short explanation inside the glossary on page 86.
- a number in square brackets represents a literature reference, which can be followed by a page number inside the referenced literature (p. 89). By example the reference “[42] (p. 43)” references literature number 42. A following page number points to the particular page *of the literature*, where the referenced information can be found – in this example on page 43.

Chapter 2: Essentials

This chapter reveals a detailed view of the Apple][computer system featuring especially the CPU as an important component for the emulation. Understanding the structure and behaviour of the CPU is fundamental for the following chapters of this thesis so all important facts are explained here. The historical context is described very shortly – see referenced literature for more historical details.

2.1 The Apple][

After a successful presentation of the Apple I computer, which was built by Steve Wozniak and published together with Steve Jobs in April 1976, the series was continued. Finally in April 1977 the Apple][was published [13] (p. 20). Together with two other home computer devices, it was the first computer system which came fully assembled. It was sold from 1977 to 1993, around 16 years, two million times [13].

Figure 1: Apple][computer with monitor at the Museum Of The Moving Image in New York City[2].

During the 16 years of production, the system was extended multiple times. Initially the Apple]["original" was released in 1977 with the Apple Integer BASIC, written by Steve Wozniak in around six weeks, missing floating point arithmetics due to time constraints [13]. After that the Apple][+ was published in 1979 with an Applesoft BASIC, written by Microsoft and including the desired floating point arithmetics [12]. Also the Disk][– the floppy disk drive – was integrated by the software, so that the system could automatically boot during startup from an inserted floppy disk. In the year 1983, the Apple //e was launched. It contained a more sophisticated graphic output, could be extended from 64K to 128K and the input of lower case characters was possible [13] (p. 41). Around the release date of this model, the Apple][c and Apple][c+ were released in 1984. It was a compact version of the Apple //e with a weight of around 4 kg

2 Image source: Marcin Wichary, CC BY 2.0, recorded on 30th december 2007, accessed on 6th june 2014, http://www.flickr.com/photos/mwichary/2151368358/.

[13] (p. 49). In 1986, the Apple IIGS was the final successor of the Apple][series, completing this successful computer series [13] (p. 57). The Apple IIGS disposed extended graphics and sound capabilities with the possibility of running original programs for the Apple][on the one hand and the graphical user interface of the Apple Macintosh series, which was started in 1984, on the other hand [13] (p. 47). The major difference to the Apple Macintosh was the fact, that the Apple IIGS provided a colored output instead of the black-and-white output of the Apple Macintosh computer until this date [13]. This last Apple][model was moreover the last model which was designed under assistance of Steve Wozniak, who stopped his active membership as a developer for Apple in february 1985 [13] (p. 51).

2.1.1 MOS Technology 6502

The main part of the Apple][computer is the 6502 microprocessor, designed by Chuck Peddle and Bill Mensch at MOS Technology [14]. As the market and interests on microprocessors started in the 70th, Intel and Motorola as leading companies developed their own microprocessors like the Intel 8080 or the Motorola 6800. All these microprocessors were full featured 8 bit machines with a wide variety of use. The prices for one of these new microprocessors were fairly high, based at around $300 for the Intel 8080 with some support chips [14].

When the 6502 was introduced, in 1975, it cost about one-sixth of the other models, available on the market. So it became the least expensive but full-featured 8 bit microprocessor one can buy these days [14]. So the 6502 was a little revolution for the microprocessor market. This was the key fact which improved the popularity of the 6502 very fast.

Technical overview

In order to get an overview over the technical features of the MOS 6502 microprocessor, bellow some important data are listed [15]:

- single +5V power supply in difference to the Intel 8080 with –5V, +5V and +12V power supply [16]
- 8 bit parallel processing with 56 instructions and basic support for pipelining
- 13 addressing modes, available for nearly all instructions
- hardware based decimal and binary arithmetics
- addressable memory up to 65K
- clock frequency between 1MHz and 2 MHz
- between two and seven machine cycles per instruction
- between one and three bytes instruction length
- →little endian byte encoding
- three hardware interrupts and one software interrupt
- register files:

- Accumulator (8 bit)
- X, Y as 8 bit index registers; used for addressing
- PC as 16 bit program counter
- SP as 16 bit stack pointer with the most significant byte hard wired to `0x01`, so that the stack is 256 byte wide from `0x01ff` to `0x0100`
- P as processor status register

Some important parts of the processor will be covered by the following sections in detail.

Memory map

The MOS 6502 processor is able of addressing 64K bytes of memory. The processor stack, interrupt vectors and the zero page are placed at determined locations [15] [17]:

- `0x0000 - 0x00ff`: **zero page**. The first 256 bytes of memory are called "zero page". These locations are used to speed up programs. As a normal addressing mode would take two bytes as instruction operands to address the whole 64K memory, it takes also many clock cycles. Using zero page addressing it is possible to speed up programs [17] (p. 61). By placing absolute addresses or even values in this first memory page, instructions only need to fetch one argument. Therefor the execution can be done faster. All other addressing modes will be covered later in section "Addressing modes" (p. 14).
- `0x0100 - 0x01ff`: **stack**. The processor stack is placed on the second memory page. This results in 256 bytes of stack size, which grows from the greatest memory location (`0x01fff`) down to the lowest and wraps around to the top if it overflows.
- `0xfffa - 0xffff`: **interrupt vectors**. These six memory locations form three 16 bit pointers into the memory. If an interrupt occurs the new program counter will be fetched from one of these locations (see p. 16)

While the zero page and interrupt vectors are intended to be overwritten and used by the application, the stack needs to be untouched for a proper code execution.

Processor status register

The status register contains seven single bit flags which are set or cleared when the instruction is executed. The following table explains the flags [15] [17]:

Bit	Flag	Description
0	C	Is set if the last instruction resulted in an overflow. It is used to perform addition and subtraction with more than one byte.
1	Z	Set if the last result was zero or equal.
2	I	If set the system will not respond to the IRQ interrupts. (The NMI cannot be masked out.)
3	D	If set the operations ADC and SBC will be set to

Bit	Flag	Description
		decimal mode, working with binary coded decimals (→BCD).
4	B	Indicates that the BRK instruction was executed. This flag cannot be changed programmatically.
5	1	Unused flag for later expansion. Mostly set to logical high level [15] [17].
6	V	Is used to indicate whether the result can be expressed in 7 bits with a sign bit.
7	N	Negative flag, indicating whether bit 7 of the result is set or not

Table 1: detailed description of the 6502 status register [15].

The flags are set by instructions and some flags can be directly set or cleared by the program. They are used directly for branch instructions and influence indirectly the executed arithmetic and logic instructions.

Addressing modes

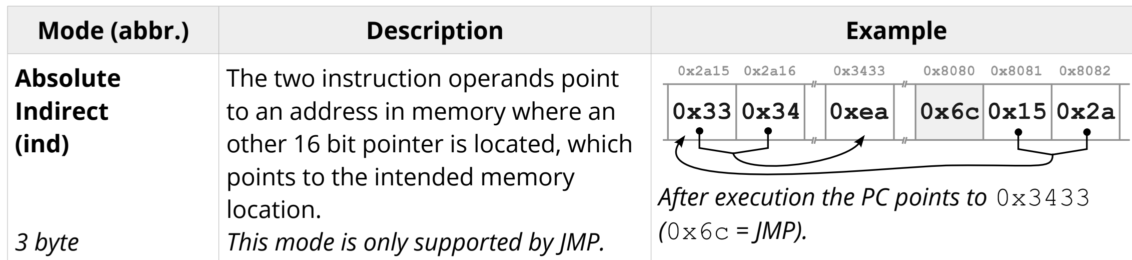
The 6502 supports thirteen different addressing modes. Not each of the 56 instructions can be used with all these addressing modes, so that 151 valid opcodes are the result. The following table explains all addressing modes [17]:

Mode (abbr.)	Description	Example
Accumulator (accu) *1 byte*	The value of the accumulator register will be used as operand.	`LDA #$15` **`ASL A`** *The accumulator register will contain the value* `0x2a` *and the carry flag is cleared.*
Implied (impl) *1 byte*	The instruction needs no value to operate with or on; e.g. the status register is used.	**`CLC`** *The carry flag is cleared.*

Relative (rel)
2 byte
	Only used with branch instructions: the 2nd byte is the branch displacement, ranging from -128 to 127. The decision of branching is made upon the state of the intended the status register flag.	`infinite_loop_1:`
		`LDA #$0`
		`BEQ infinite_loop_1`
		If zero is loaded the zero flag is set. A Branch will be executed.

Immediate (imm) *2 byte*	The operand is located in the 2nd byte of the instruction.	`LDA #$42` **`AND #$2a`** *The accumulator register will contain the value* `0x02`.

Mode (abbr.)	Description	Example
Indexed Indirect (indX) *2 byte*	The sum of the second byte and the value of the X register point to a memory location in the zero page where a 16 bit address is located, which points to the final location. Wraparound is used to stay inside the zero page.	*The value of the accumulator will be stored to* `0x3230` *(*`0x81 0x03` *= STA ($03,X)).*
Indirect Indexed (indY) *2 byte*	The second byte points into the zero page where a 16 bit pointer is located. The sum of the 16 bit pointer and the value of the Y register point to the intended memory location.	*The value of the accumulator will be stored to* `0x3230` *(*`0x91 0x04` *= STA (0x04),Y).*
Zero Page (zp) *2 byte*	Allows working with an operand in the zero page by only specifying the lower byte of the memory location pointer. The upper byte is `0x00`. This saves one instruction operand and also execution cycles.	*After execution* `0x0015` *contains* `0x2a` *(*`0xe6 0x15` *= INC $15).*
Zero Page X (zpX) *2 byte*	Points to a memory value inside the zero page. The pointer is the sum of the second instruction byte and the X register. The sum wraps around at `0x00ff` to remain in the zero page.	*After execution* `0x0015` *contains* `0x2a` *(*`0xe6 0x15` *= INC $15, X).*
Zero Page Y (zpY) *2 byte*	Same behaviour as Zero Page X addressing mode, but with Y register instead of X. *Supported by: LDX & STX.*	*See Zero Page X with Y register.*
Absolute (abs) *3 byte*	The second and third byte specify an absolute memory address, lower byte first. Thus, the complete 64K memory is addressable.	*After execution* `0x2a15` *contains* `0x42` *(*`0xee 0x15 0x2a` *= INC $2a15)*
Absolute X (absX) *3 byte*	The second and third byte point to a memory location in the 64K memory. The final pointer consists of the sum of the 16 bit pointer and the value of the X register.	*After execution* `0x2a15` *contains* `0x42` *(*`0xfe 0x11 0x2a` *= INC $2a15, X).*
Absolute Y (absY) *3 byte*	Same behaviour as “Absolute X” addressing mode, but with the Y register instead of X.	*See Absolute X with Y register.*



Mode (abbr.)	Description	Example
Absolute Indirect (ind) *3 byte*	The two instruction operands point to an address in memory where an other 16 bit pointer is located, which points to the intended memory location. *This mode is only supported by JMP.*	0x2a15 0x2a16 0x3433 0x8080 0x8081 0x8082 0x33 0x34 0xea 0x6c 0x15 0x2a *After execution the PC points to* `0x3433` *(*`0x6c` *= JMP).*

Table 2: detailed explanation of all different addressing modes supported by the MOS 6502 microprocessor.

Please note that the explanations in table 2 assume that an instruction consists at least of one byte, the instruction opcode itself. The following two operands or arguments are optional.

Instruction overview

Table 3 (p. 17) gives a simple outline of all the instruction opcodes available on the MOS 6502 microprocessor along with their specifications (addressing mode, length in byte, cycle count).

In every cell of the table the mnemonic of the instruction is displayed, followed by the addressing mode from table 2 and finally followed by the instruction length in bytes and the number of cycles this instruction takes. Empty fields are “illegal” opcodes.

As indicated by the asterisks, there are special rules for the number of cycles, which are taken by specific instructions:

- * all cycle times labeled with a single asterisk take one more clock cycle if a page boundary in memory is crossed. This means if the final composed memory address crosses a page boundary, e.g. low byte on `0x2aff` and high byte on `0x2b00`, the additional cycle is added.
- ** (only branch instructions) since a branch has several possibilities to perform, there are different cycle consumptions: a branch not taken requires the standard two cycles. If the branch is taken, it requires three clock cycles. And if it is taken and crosses a page boundary, it requires four cycles.

The remaining 105 instructions – represented by an empty cell in table 3 – are not explained in this document but they have some function, indeed. The problem here is that they are “illegal”. The manufacturer has not given any sense to this instructions and also not tied them to something like a “no operation”. Instead they all do some fancy things on a real MOS 6502 microprocessor. Some illegal instructions perform stable operations, like a double `NOP`. Others have an unpredictable behaviour, which can freeze the microprocessor. Hobbyist carried out tables of illegal instructions that seem to perform stable actions. Despite this, they are not covered in the official documents and also not in this document.

Interrupts

The interrupt feature is used to let the processor suspend the current sequential execution and jump to another code fragment. This interrupt event can be triggered by hardware and software and needs immediate attention. By example a keyboard input, which needs to be processed.

	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
0x00	**BRK** (impl) 1 / 7	**ORA** (indX) 2 / 6				**ORA** (zp) 2 / 3	**ASL** (zp) 2 / 5		**PHP** (impl) 1 / 3	**ORA** (imm) 2 / 2	**ASL** (accu) 1 / 2			**ORA** (abs) 3 / 4	**ASL** (abs) 3 / 6	
0x10	**BPL** (rel) 2 / 2**	**ORA** (indY) 2 / 5*				**ORA** (zpX) 2 / 4	**ASL** (zpX) 2 / 6		**CLC** (impl) 1 / 2	**ORA** (absY) 3 / 4*				**ORA** (absX) 3 / 4*	**ASL** (absX) 3 / 7	
0x20	**JSR** (abs) 3 / 6	**AND** (indX) 2 / 6			**BIT** (zp) 2 / 3	**AND** (zp) 2 / 3	**ROL** (zp) 2 / 5		**PLP** (impl) 1 / 4	**AND** (imm) 2 / 2	**ROL** (accu) 1 / 2		**BIT** (abs) 3 / 4	**AND** (abs) 3 / 4	**ROL** (abs) 3 / 6	
0x30	**BMI** (rel) 2 / 2**	**AND** (indY) 2 / 5*				**AND** (zpX) 2 / 4	**ROL** (zpX) 2 / 6		**SEC** (impl) 1 / 2	**AND** (absY) 3 / 4*				**AND** (absX) 3 / 4*	**ROL** (absX) 3 / 7	
0x40	**RTI** (impl) 1 / 6	**EOR** (indX) 2 / 6				**EOR** (zp) 2 / 3	**LSR** (zp) 2 / 5		**PHA** (impl) 1 / 3	**EOR** (imm) 2 / 2	**LSR** (accu) 1 / 2		**JMP** (abs) 3 / 3	**EOR** (abs) 3 / 4	**LSR** (abs) 3 / 6	
0x50	**BVC** (rel) 2 / 2**	**EOR** (indY) 2 / 5*				**EOR** (zpX) 2 / 4	**LSR** (zpX) 2 / 6		**CLI** (impl) 1 / 2	**EOR** (absY) 3 / 4*				**EOR** (absX) 3 / 4*	**LSR** (absX) 3 / 7	
0x60	**RTS** (impl) 1 / 6	**ADC** (indX) 2 / 6				**ADC** (zp) 2 / 3	**ROR** (zp) 2 / 5		**PLA** (impl) 1 / 4	**ADC** (imm) 2 / 2	**ROR** (accu) 1 / 2		**JMP** (ind) 3 / 5	**ADC** (abs) 3 / 4	**ROR** (abs) 3 / 6	
0x70	**BVS** (rel) 2 / 2**	**ADC** (indY) 2 / 5*				**ADC** (zpX) 2 / 4	**ROR** (zpX) 2 / 6		**SEI** (impl) 1 / 2	**ADC** (absY) 3 / 4*				**ADC** (absX) 3 / 4*	**ROR** (absX) 3 / 7	
0x80		**STA** (indX) 2 / 6			**STY** (zp) 2 / 3	**STA** (zp) 2 / 3	**STX** (zp) 2 / 3		**DEY** (impl) 1 / 2		**TXA** (impl) 1 / 2		**STY** (abs) 3 / 4	**STA** (abs) 3 / 4	**STX** (abs) 3 / 4	
0x90	**BCC** (rel) 2 / 2**	**STA** (indY) 2 / 6			**STY** (zpX) 2 / 4	**STA** (zpX) 2 / 4	**STX** (zpY) 2 / 4		**TYA** (impl) 1 / 2	**STA** (absY) 3 / 5	**TXS** (impl) 1 / 2			**STA** (absX) 3 / 5		
0xa0	**LDY** (imm) 2 / 2	**LDA** (indX) 2 / 6	**LDX** (imm) 2 / 2		**LDY** (zp) 2 / 3	**LDA** (zp) 2 / 3	**LDX** (zp) 2 / 3		**TAY** (impl) 1 / 2	**LDA** (imm) 2 / 2	**TAX** (impl) 1 / 2		**LDY** (abs) 3 / 4	**LDA** (abs) 3 / 4	**LDX** (abs) 3 / 4	
0xb0	**BCS** (rel) 2 / 2**	**LDA** (indY) 2 / 5*			**LDY** (zpX) 2 / 4	**LDA** (zpX) 2 / 4	**LDX** (zpY) 2 / 4		**CLV** (impl) 1 / 2	**LDA** (absY) 3 / 4*	**TSX** (impl) 1 / 2		**LDY** (absX) 3 / 4*	**LDA** (absX) 3 / 4*	**LDX** (absY) 3 / 4+	
0xc0	**CPY** (imm) 2 / 2	**CMP** (indX) 2 / 6			**CPY** (zp) 2 / 3	**CMP** (zp) 2 / 3	**DEC** (zp) 2 / 5		**INY** (impl) 1 / 2	**CMP** (imm) 2 / 2	**DEX** (impl) 1 / 2		**CPY** (abs) 3 / 4	**CMP** (abs) 3 / 4	**DEC** (abs) 3 / 6	
0xd0	**BNE** (rel) 2 / 2**	**CMP** (indY) 2 / 5*				**CMP** (zpX) 2 / 4	**DEC** (zpX) 2 / 6		**CLD** (impl) 1 / 2	**CMP** (absY) 3 / 4*				**CMP** (absX) 3 / 4*	**DEC** (absX) 3 / 7	
0xe0	**CPX** (imm) 2 / 2	**SBC** (indX) 2 / 6			**CPX** (zp) 2 / 3	**SBC** (zp) 2 / 3	**INC** (zp) 2 / 5		**INX** (impl) 1 / 2	**SBC** (imm) 2 / 2	**NOP** (impl) 1 / 2		**CPX** (abs) 3 / 4	**SBC** (abs) 3 / 4	**INC** (abs) 3 / 6	
0xf0	**BEQ** (rel) 2 / 2**	**SBC** (indY) 2 / 5*				**SBC** (zpX) 2 / 4	**INC** (zpX) 2 / 6		**SED** (impl) 1 / 2	**SBC** (absY) 3 / 4*				**SBC** (absX) 3 / 4*	**INC** (absX) 3 / 7	
	0x00	**0x01**	**0x02**	**0x03**	**0x04**	**0x05**	**0x06**	**0x07**	**0x08**	**0x09**	**0x0a**	**0x0b**	**0x0c**	**0x0d**	**0x0e**	**0x0f**

Table 3: outline of the complete instruction set of the MOS 6502 microprocessor.

The 6502 microprocessor has hardware and software interrupt features. If such an interrupt occurs the new program counter will be loaded from a “vector” address. Those are two memory locations forming a 16 bit memory value to address the complete 64K memory. It supports the following interrupts [15] [17]:

- **IRQ:** this maskable interrupt is hardware generated by a level change on the IRQ pin [18]. If the current instruction is executed, the processor will load the new program counter from the locations `0xfffe` and `0xffff` and continue execution from there. If the “I” flag in the status register is set, the interrupt will be ignored. Also the two bytes of the program counter and the current processor status register are pushed onto the stack.
- **BRK:** is the same as an IRQ with the difference that it is triggered by software by the `BRK` instruction. In this case also the “B” flag in the status register will be set by this interrupt. This pushes also the two bytes of the program counter and the processor status register onto the stack. It is intended to use for debug purposes or as →system call feature.
- **NMI:** this is called the “non-maskable interrupt”, because it cannot be masked out and occurs every time triggered. It is also triggered by the hardware NMI pin, when it changes from high to low [18]. Then the processor loads the new program counter from `0xfffa` and `0xfffb`. This pushes also the two bytes of the program counter and the processor status register onto the stack.
- **RESET:** also an hardware interrupt, triggered by the RESET pin. This interrupt resets the stack pointer to `0xfd`[3] and loads the new program counter from `0xfffc` and `0xfffd`.

As the NMI interrupt cannot be turned of, it can interrupt a normal IRQ or BRK interrupt. This kind of interrupt was created for external devices that cannot even wait until the interrupt execution is finished, e.g. HSYNC or VSYNC of a →CRT signal, which needs very proper timing.

The RESET interrupt is the strongest interrupt, causing the CPU to continue execution on the fetched address from the RESET vector. It resets the stack, so that one cannot return to the previous execution location by `RTI`.

Issues on the 6502 and further decisions

The fact that the MOS 6502 microprocessor is running until now in many devices properly does not imply that its free of mistakes or issues. There are many issues filed. Instead of listing all issues, only those which are important for this thesis are listed here:

- illegal opcodes: as mentioned before, the unused opcodes are not tied to something “neutral” in the original MOS 6502 microprocessor variant (later variants solved this issue). Because of the fact that most of them are undocumented and uncontrollable, the following decision is made: *only the by the*

3 The SP gets reset to `0xfd`, because the 6502 injects a `BRK` instruction on RESET. While executing the `BRK` instruction it writes the two PC bytes and the processor status register onto the stack. In case of RESET the actual data is not needed so the control lines change to read while `BRK` tries to write onto the stack and no data is written onto the stack. Only the stack pointer decremented by three from `0xff` [19] [20].

manual [17] and datasheet [15] specified and legal set of instructions is covered by this thesis (as seen in table 3).

- “trouble” with JMP: the JMP instruction in indirect addressing mode has a bug. If the lower byte of the new program counter is on the last location of a page (e.g. `0x00ff`), the next location is not the first location of the new page (e.g. `0x0100`), the 6502 takes the first location of the current page (e.g. `0x0001`) instead of the new page. *This “mistake” will be considered in the emulator implementation.*

2.1.2 Memory organisation

After explaining the key component – the 6502 microprocessor – one can move on with the Apple][system. Due to the fact that the processor can address 64K of memory, the Apple][could use up to 64K memory. The memory is further divided into two sections:

- the read and write section for user program space, from `0x0000` to `0xbfff`. This are 48K of total memory available to the user (including the zero page and stack).
- a section for memory mapped I/O and expansion ROM from `0xc000` to `0xcfff`.
- the read only section stored in ROM modules from `0xd000` to `0xffff`. These 12K bytes of memory are reserved for the monitor program and the Integer BASIC. The monitor program is a masterful program [21] (p. 40) that provides “system call” functions and controls the programs. The Integer BASIC is the “operating system” for the user providing a BASIC command line.

The following table 4 takes a detailed view on the memory showing a total overview in a graphical notation.

<table>
<tr><th>Page(s)</th><th>Size in bytes</th><th colspan="2">Intended function by the Apple][</th></tr>
<tr><td>0x00</td><td>256</td><td colspan="2">Zero page for 6502 microprocessor (acceleration)</td></tr>
<tr><td>0x01</td><td>256</td><td colspan="2">6502 microprocessor stack</td></tr>
<tr><td>0x02</td><td>256</td><td colspan="2">GETLN input buffer (Monitor program / BASIC)</td></tr>
<tr><td>0x03</td><td>256</td><td colspan="2">Other monitor vector locations</td></tr>
<tr><td>0x04 … 0x07</td><td>1,024</td><td colspan="2">Primary “page” for text and low resolution (LoRes) graphics (see “LoRes graphics mode“, p. 23)</td></tr>
<tr><td>0x08 … 0x0b</td><td>1,024</td><td>Secondary page for text and LoRes graphics (double-buffer)</td><td rowspan="5">Free to use</td></tr>
<tr><td>0x0c … 0x1f</td><td>5,120</td><td></td></tr>
<tr><td>0x20 … 0x3f</td><td>8,192</td><td>Primary page for high resolution (HiRes) graphics (see “HiRes graphics mode“, p. 23)</td></tr>
<tr><td>0x40 … 0x5f</td><td>8,192</td><td>Secondary page for HiRes graphics (double-buffer)</td></tr>
<tr><td>0x60 … 0xbf</td><td>24,576</td><td></td></tr>
</table>

Page(s)	Size in bytes	Intended function by the Apple][	
`0xc0 … 0xcf`	4,096	Memory mapped I/O (no physical memory at all): • important I/O locations at the beginning (e.g. graphics mode) • peripheral card I/O space • peripheral card ROM space • expansion ROM	
`0xd0 … 0xf7`	10,240	BASIC language interpreter	ROM
`0xf8 … 0xff`	2,048	Monitor program (inc. 6502 interrupt vectors)	
	65,536	**# of total memory locations**	

Table 4: Apple][memory map, all 256 pages and their function (if any) [21] (p. 69).

The pages `0xc0` to `0xcf` perform an important action: the memory mapped I/O. That means that this memory locations have no physical memory at all. Instead, the address on the address bus is used to trigger some kind of action. Mostly these locations are read and the result is an undefined value which is discarded, because referencing the specific memory location has performed the desired hardware action (e.g. beep with the speaker).

The memory mapped I/O section plays an important role. The following table outlines the general structure of this memory region:

	0x0	**0x1**	**0x2**	**0x3**	**0x4**	**0x5**	**0x6**	**0x7**	**0x8**	**0x9**	**0xa**	**0xb**	**0xc**	**0xd**	**0xe**	**0xf**
0xc000	Keyboard data (see “2.1.4 The keyboard“, p. 24)															
0xc010	Keyboard clear (see “2.1.4 The keyboard“, p. 24)															
0xc020	Cassette output toggle															
0xc030	Speaker toggle															
0xc040	Utility strobe															
0xc050	gr	tx	no	mix	pri	sec	lore	hire	an0		an1		an2		an3	
0xc060	Game controller input, cassette input								*0xc060 – 0xc067 mirrored*							
0xc070	Game controller strobe															

Table 5: important built-in I/O locations [21] (p. 79).

As one can see, these locations manage important I/O functions like the keyboard input or (sound) speaker output and other I/O related stuff. The 16 memory locations, beginning at `0xc050`, manage the graphic output of the entire Apple][. The abbreviations have the following meanings:

- “gr” and “tx” stand for “graphics mode” and “text mode” and toggle between fullscreen graphics and fullscreen character display.
- “no” and “mix” decide whether the screen is “not mixed” or “mixed”. In mixed mode together with graphics mode, the Apple][displays four lines of text mode on the lower section of the screen (e.g. to see graphics while entering BASIC commands).

- “pri” and “sec” decide which location for video data in memory is used – the primary or the secondary. This can be used for frame double-buffering.
- “lore” and “hire” distinguish between the low resolution mode and high resolution mode.
- “an0” to “an3” are called “annunciator ports”. These are simply digital output lines which can be turned off by referencing the lower address and on by referencing the higher address. This feature is useful to communicate to a serial device or something else.

Furthermore there is a difference between this memory locations. They are either a “toggle switch” or a “soft switch”.

These toggle switches are internally flip-flops. Each time a read occurs the flip flop will change its state to the opposite (logical low or high). By reading this memory location the flip flow will be toggled. By writing to this location, the flip flop will be toggled twice and is finally in the same state as it was before. An example of a toggle switch is the speaker output. By reading the location it will “click”. Performing this action faster and cyclic, a tone might be generated [21] (p. 20, p. 79).

Soft switches on the other hand are as simple as light switches with two states: every state has a memory location. If this location is invoked by a read or write[4], the switch is thrown to this state [21] (p. 79). The video mode selection or annunciator ports are made out of soft switches: if a read occurs on `0xc050`, the Apple][output is forced to fullscreen graphics output. If `0xc051` is referenced, the Apple][is forced to fullscreen text output.

The Apple][had also eight extension card slots and the possibility to map their I/O registers and ROM code into the main memory (table 4). This enabled the first Apple][to use a disk drive although it was not yet invented when it was released. The disk drive came with an extension card, which was inserted in one of these slots. The driver firmware was mapped into the main memory place for peripheral cards ROM[5] and the user was able to work with the disk drive. As *important and interesting these features might be they are not covered deeper on this thesis, because they aren't implemented yet. An implementation would go beyond the scope of this thesis.*

As seen before, the user could work with their programs in a maximum of 48K bytes of memory. In those early days this “huge” amount of memory was very expensive. So the original Apple][was shipped containing 4K bytes of user memory [21] (p. 71). The motherboard contained 24 + 1 sockets for ➝DIP RAM modules. The 24 sockets are split into tree “banks” or rows of eight sockets holding a static RAM ➝IC and forming a byte, because a static RAM ➝IC stored only bits, not bytes. The last socket was filled with jumpers to place the memory of the eight sockets inside the main memory at the right location. Every group of eight slots needs to contain the same memory modules, if used, to work properly [21] (p. 71).

With this flexible system and RAM modules available in 4 and 16 Kbit[6], there are nine

4 E.g. by using an assembler instruction onto this memory location, which will read it or write to it.

5 From `0xc100` to `0xc7ff` every peripheral card slot has 256 byte PROM (program ROM, e.g. the “driver”) space and a card could request 2K expansion ROM from `0xc800` to `0xcfff` [21].

6 A module of 4 Kbit memory contains 4.096 distinct memory locations of one bit length (not one byte!). This is the reason, why they are grouped by eight: eight of these modules form a byte of data in the memory.

valid memory combinations: 4K, 8K, 12K, 16K, 20K, 24K, 32K, 36K and 48K [21] (p. 71). Inserting the RAM modules into the 24 sockets and setting the jumpers correctly enabled the user to extend the memory later on, since memory ICs were expensive in those days.

2.1.3 Video output

As described, the video RAM is mapped into the memory and it supports different types of video output:

1. **text mode –** the screen is filled with 960 characters.
2. **low resolution (LoRes) graphics mode –** the screen is filled with 1.920 colored blocks. Every block can have one of 16 colors of the Apple][color palette.
3. **high resolution (HiRes) graphics mode –** the screen can display a 280 by 192 pixels wide image with 53.760 distinct dots. Every dot could possibly have one color out of six. But there are some special rules restricting this.
4. **mixed mode –** in this mode one of the graphics modes is enabled and the lower four lines of the screen are set to text mode, e.g. showing the BASIC command line.

Text mode

In the text mode the screen is filled with characters in 40 columns over 24 rows. Every character represents one memory value at the LoRes graphics memory page from `0x0400` – `0x7fff` (or `0x800` – `0x0bff` if secondary page selected).

A character consists of a 5 x 7 dot graphics with an one dot wide space on the left and the right and an one dot high line above every character to separate the lines. So every character is 7 x 8 dots in size. The character set consists of 64 characters:

	0x0	**0x1**	**0x2**	**0x3**	**0x4**	**0x5**	**0x6**	**0x7**	**0x8**	**0x9**	**0xa**	**0xb**	**0xc**	**0xd**	**0xe**	**0xf**
0x0	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0x1	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0x2		!	“	#	$	%	&	'	(	)	*	+	,	-	.	/
0x3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?

Table 6: the Apple][character set [22] (figure 8.4, chapter 8).

Each character has one byte of data and so it can represent 256 distinct symbols. The character set with its 64 characters is repeated four times inside this 256 distinct symbols. This enables different kinds of displays for every character set block:

- `0x00` – `0x3f` (`0b00******`): the characters are displayed inverse (black characters on white background)
- `0x40` – `0x7f` (`0b01******`): the characters are flashing. They are changing fast from inverse to normal display.
- `0x80` – `0xbf` (`0b10******`): normal character display (white on black).
- `0xc0` – `0xff` (`0b11******`): repetition of the previous block.

The memory layout on the display page is quite complex and fragmented – the character rows have free memory locations as spacers. *In order to stay in the scope of this document, this is not covered detailed. More information can be found at [21] (p. 18, fig. 2).*

LoRes graphics mode

In the low resolution mode the screen is filled with 1,920 colored blocks, each is 7 x 4 pixels in size. This leads to a resolution of 40 x 48 blocks. Every block can have one of 16 colors.

It is easy to see that every text character consists of two blocks. So the lower →nibble of the character is responsible for the upper block and the upper nibble for the lower block. These two nibbles form the corresponding character byte. Because of this fact the memory and memory layout for the text mode can be used to display LoRes graphics, with only changing the interpretation of the values by toggling the soft-switch in `0xc050`.

Figure 2: color pallet of the Apple][low resolution graphics mode [23].

HiRes graphics mode

The high resolution mode enables the user to display a 280 by 192 dots wide "image" with 53,760 distinct dots. Every dot can have theoretically one color of black, white, violet, green, red and blue. The two memory pages for primary and secondary HiRes graphics are both 8K long and located as shown in table 4 (p. 20). If the Apple][has less than 16K memory, this graphic mode cannot be used because of missing memory.

The coloring of the dots is fixed by their position on the screen. The background or the unset dots are always colored in black. Each bit of a byte in the memory represents one dot. The eight bit is not displayed, but used to switch between the colors for the dots. Every line of dots is drawn from left to right going through the bytes from the first (least significant) bit to the seventh bit [21] (p. 19). On black-and-white monitors every set bit is displayed white, otherwise it is displayed in the background color black. The "rules" for coloring are [21] (p. 19):

1. If the dot is not set, it is displayed ***black***.
2. If the dot is set and in an even numbered dot column (0, 2, 4, 6, ..., 278), it is displayed in the color ***violet***.
3. If the dot is set and in an odd numbered dot column (1, 3, 5, ..., 279), it is displayed in the color ***green***.
4. If two dots are placed side-by-side, both are displayed ***white***.
5. If bit seven of the byte is turned on, the colors ***violet/green*** are replaced by the colors ***blue/red***.

2.1.4 The keyboard

An other important component of the Apple][is its keyboard, which is built-in into the enclosure. The specifications of the keyboard are [21] (p. 6):

- 52-key typewriter-like keyboard, supporting only uppercase characters.
- supports up to two key pressed at the same time.
- a key produces only a key code when it is pressed down. During the key is held down, no further key code is generated.
- special keys:
  - **CTRL, SHIFT**: they generate no key codes by them self, but in combination with other keys (e.g. “SHIFT + 1 = !” or “CTRL + G = BELL”).
  - **REPT**: due to the fact that a pressed key only produces one character independently from the time it is pressed, this key provides a multi keypress feature. If the REPT key is pressed alone, the last generated character is reproduced once. If the REPT key is pressed together with an other key the key is reproduced at about ten times per second [21] (p. 7). If one of the two keys is released, the process is stopped.
  - **RESET**: this key is directly connected to the the MOS 6502 microcontroller RESET pin. If this key is pressed, the RESET pin at the microprocessor is pulled to ground and goes high on release[15] (p.8). On this positive edge a RESET hardware interrupt is triggered. See section “Interrupts“ (p. 16) for details.
  - **ESC, ←, →**: keys to provide special functions like an “ESCape” action or moving the cursor.
- memory mapped I/O locations to access the key code programmatically: `0xc00*` (data), `0xc01*` (clear)

Figure 3: sketch of the Apple][original keyboard with all its 52 keys (the power lamp at the lower left corner not included) [21] (p. 6).

If a key is pressed, the generated key code is placed in the memory location `0xc00*` and it is greater than 128[7]. The key code will remain there until another key is pressed or it is cleared. By referencing the memory location `0xc01*` the value in `0xc00*` gets subtracted by 128. Every value beneath 128 is assumed as no key code. But one can still recover the last pressed key code by reading the value and adding 128 to it.

7 Bit seven is set in the Apple][character set in contrast to the →ASCII standard where the key code needs only the first six bits.

2.1.5 Other hardware features

There are a lot more hardware features of the Apple][, which would go beyond the scope of this document. So they will be summarized very shortly below:

- as mentioned before, the Apple][contains eight slots for extension cards. All address lines and other important processor lines are available to this cards. This enabled the Apple][to be extended later on. There are various hardware extension "cards" available. Some examples are [21]:
  - extension of the 40 by 24 text screen to 80 by 48 characters, with a card providing more text memory and the needed firmware.
  - extension of the Apple][by a disk drive: the extension card is connected to the disk drive and provides the firmware to run it.
  - or nowadays: connection of a ➝Raspberry Pi with an extension card adapter to extend the Apple][memory or use other features provided by the Raspberry Pi (HDMI output etc.) [24].
- the speaker is a simple 8Ω speaker, connected to the internal electronics. A reference of the memory location `0xc03*` allows to change the voltage level from low to high or vice versa and one can hear a click. Performing this action with a correct timing and duration will produce a sound.
- The Apple][had three digital input pins and four analog input pins which could be read by software as general purpose input. The four annunciator ports and a utility strobe represent the general purpose output. The strobe is a simple output which will drop from +5V to 0V for 0.5 μs by referencing a memory location (like the speaker) [21] (p. 20).
- The Apple][could also be interfaced by a simple analog game controller called "paddle". They input data into the system by using the plain analog and digital input pins, which were available on the motherboard [21] (p. 24 and 78).
- The Apple][original was designed to use a regular cassette and cassette recorder to store data permanently. For this it has two audio jacks: one for storing (microphone-in of the cassette recorder) and the other for reading (earphone-out of the cassette recorder). The system Monitor program provides all firmware to save or to load from this medium binary data. In the I/O memory two memory locations (`0xc02*` and `0xc060`) are dedicated to control an electronic circuitry which interprets the "sound" input of a cassette or generates "sound" output to store [21] (p. 22 and 79).

2.1.6 Software insights: System Monitor & BASIC

The system monitor is a masterful program providing "system call" functions and some limited input prompt. The original Apple][started the System Monitor program after power up.

The Monitor prompt starts with an asterisk ("`*`"). From System Monitor one can get into BASIC using the key stroke "CTRL + B" and "ENTER". From BASIC one can get into the

Monitor program by entering “`CALL -151`”. It provides an input prompt for advanced programmers with actions like: examine, change, move and verify memory or a mini assembler [25] (pp. 68).

Using this “mini-assembler”, one is able to write MOS 6502 assembler programs in an enhanced environment, which provides helping resources. There is also the possibility to save the data to the cassette or read from a cassette. The entire system Monitor is located in the upper 2K of ROM.

Beneath this 2K of System Monitor, there are 10K of ROM dedicated to the current BASIC language: on the Apple][“original“ the Integer BASIC and the Applesoft BASIC on the Apple][plus [21].

The Integer BASIC was written by Steve Wozniak in a very short amount of time. It leaks floating point arithmetics but is quite faster than the Applesoft BASIC. One is using the Integer BASIC, if the prompt character is an rightwards arrow (“>”). Because of its speed benefit against the Applesoft BASIC it is more popular for games, because games only need integer values [13] [12].

The Applesoft BASIC was written by Microsoft, because Steve Wozniak was occupied with developing the Apple Disk][and the customers of the Apple][requested floating point arithmetics [13] [12]. The Applesoft BASIC is slower than the Integer BASIC. One can recognize that the current running BASIC is the Applesoft by looking at the prompt character: it is a right square bracket (“]”).

2.2 Microcontrollers vs. Microprocessors

There are devices called “microprocessors” (or CPU or processor). This devices are multi purpose devices, open to compute in an infinite variety of different tasks. It accepts digital input data, processes it according to its instructions and outputs the result [26]. With only a microprocessor, a computer is useless. There is the need for plenty of peripheral devices like main memory (aka RAM), a hard disk, a graphics card, input controllers, for keyboard and mouse devices, and other components to create an environment like a standard desktop personal computer (PC), which can feed program instructions and data into the microprocessor and use the result. All these components together form a (modern) personal computer [26]. As seen before, the Apple][had all theses modules, forming an – for this time – incredible computer (system).

In opposite to that, a microcontroller is a quite smaller device and does not need any peripheral components. It consists of →SRAM, →Flash memory, →EEPROM memory and a CPU. So it is an entire “personal computer” in a much smaller package, but also with less performance in it.

The key difference is, that the microcontroller has a defined relationship between input and output and is programmed once with a software for a specific task and then build into an electrical circuit for this fixed task. Since it is responsible only for a very specific task, it does not need much computational power. This results in a small device with less peripherals and also less power consumption, whereby a microprocessor can do a wide variety of different tasks, for example running different software on a computer, but requires peripherals to interact [27].

2.3 Different architectures

Beyond the difference of the technical specifications and different purposes of micro-controllers and microprocessors most microcontrollers make use of a different overall system architecture. The two general architecture styles are explained in the following.

2.3.1 Von Neumann architecture

A normal personal computer, equipped with a microprocessor, is build upon the "Von Neumann" system architecture, labeled after the mathematician and physicist *John von Neumann*. He proposed that data and program instructions should retain both in the same memory and processed by a processor with an ALU and registers – a microprocessor [28].

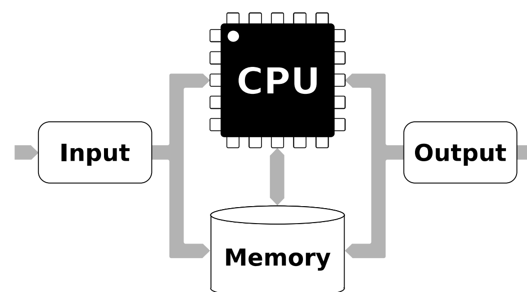


Figure 4: sketch of the Von Neumann architecture. The "Memory" stores data and programs and the CPU works on this data.

This is a simple model for a computer, because it involves only one data bus, to transfer program and data from RAM or secondary storage to the CPU and get the result back into storage. It also enables the system to load data from a secondary storage into memory and then treats it as program data. This kind of feature is used every time a computer is started: it loads the operating system from hard drive and then executes it. This kind of architecture allows further other enhancements like just-in-time optimization, where the instructions get optimized during runtime and slower code is replaced. All this features are only possible if data and program data reside in the same memory [29].

The disadvantages of this architecture are that data memory regions could get executed due to an error or malware program, which results in uncontrollable or undesired behaviour and maybe in data loss [29]. Another great disadvantage is known as the "Von Neumann bottleneck": data and program data are fetched through the same memory bus. So they cannot be fetched at the same time, which slows down the speed of the CPU [28].

2.3.2 Harvard architecture

The counterpart is the Harvard architecture. In this architecture, the data memory is strictly separated from the program memory. With the term "Harvard architecture" the "modified Harvard architecture" is meant [30] [31]. It softens the strict initial definition and allows to access the program memory to provide the possibility of loading program data [31].

The main unique characteristics of this kind of architecture are the separated busses for data and program memory and separated memory modules respectively. This feature solves the “Von Neumann bottleneck” and allows fetching data while fetching program instructions at the same time [30]. This feature improves the throughput. It also enhances the security, since there is no direct interconnection and it is harder to execute program data.

There are not many microprocessors which are build upon Harvard architecture, due to the fact that making it useable and writing programs for it is more complicated due to the disjoint busses. Many microcontrollers are made upon this architecture, especially the Atmel AVR microcontroller series, which is used for implementation later on in this thesis.

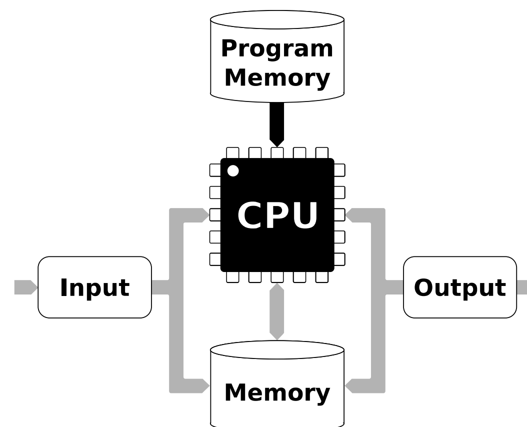


Figure 5: sketch of the Harvard architecture: the "Memory" is only used for data and the "Program Memory" stores only programs. They are not connected.

An Atmel AVR microcontroller comes with a CPU, a flash memory, an SRAM and an EEPROM. The Flash memory stores the program data and is non-volatile. The SRAM (volatile) is intended to store program data (variables, strings etc.) during runtime. The EEPROM (non-volatile) is a slow secondary memory, separated from the flash memory and dedicated to permanent data storage. So there are existing two independent busses for data and program memory.

That this architecture follows the “modified Harvard architecture” can be seen by the fact that there are assembler instructions like “`lpm`” and “`spm`” to load and store data into or from program memory [31].

Chapter 3: Software implementation

With all this base knowledge collected in the last chapter, this chapter will move on to the actual implementation of the emulator and explain the idea behind the software implementation.

3.1 Concept and basic setup

The goal of this chapter is to convey the structure of the software implementation in order to reach the target of a portable handheld emulator of the Apple][computer system. Due to the fact that the emulation of the MOS 6502 microprocessor plays a key role in emulating the Apple][, this part of the software is discussed very deep to ensure that the implementation idea is expounded well enough. Other parts maybe discussed only superficial without going into depth to stay in the scope of this document in means of time and length.

The code shown in this chapter is always very short and abstract. Its target is to show the structural thoughts and concepts and not the whole implementation. If there is the need for further information about the code, one can take a look into the source code. It is documented very well and might answer open questions.

Hardware and software share a really tight relationship. These topics were split into two distinct chapters in order to keep the overview and explain details of both components structured. Through the tight relationship it cannot be avoided that the following hardware chapter is referred sometimes, because of the fact that the hardware restricts the software and arises the need for further tweaking of the software. Most details of the hardware are hidden in this chapter and the focus is set onto the software side of implementation.

Technical software implementation details

The tools used to implement the software emulator part were AVR Studio 4.19 along with the latest WinAVR[8] toolchain containing `avr-gcc` in version 4.3.3. Because of the fact that this is a pretty old version of `avr-gcc`, the other parts of the project were implemented using the AVR toolchain for Mac OS X 10.9 containing the most recent `avr-gcc` 4.8.2.

8 Used version of WinAVR was 20100110 (still latest for this platform).

Further any speed tests which are made using an Arduino, use the Arduino Uno board in revision 3 together with the Arduino IDE 1.0.5. Detailed information on the speed test setup can be seen in section "6.3 “Speed” measurement setup“ (p. 91).

Hardware limitations

Due to the fact that the target is a complete self-contained Apple][emulator on a **single** AVR microcontroller there are some hardware limitations which will also limit and influence the software implementation.

- As we will see, speed is an important factor for the emulator. The more, the better. The used microcontroller will support a clock speed up to 20 MHz. So 20 MHz are used.
- The Apple][had up to 64K main memory. 16K were used for ROM and memory mapped I/O, leaving 48K user space. Not every Apple][model had 48K of memory installed. By example the original Apple][came with 4K of 48K user space. By replacing the main memory with more memory, the owner was able to increase the size of the main memory (see “2.1.2 Memory organisation”, p. 19). Because of the fact that the used microcontroller provides only 16K of SRAM, the Apple][will be realized in a configuration level of 12K.
- Because of only having 12K out of 48K possible user space, the high resolution mode is not available and not implemented.
- The demand profile for this thesis (see section “Demand profile”, p. 7) masks out the BCD mode of the 6502 CPU. It is very complex and to the knowledge of the author not used in the major Apple][software (or even nowadays).

3.2 Emulation of the MOS 6502 CPU

The emulation of the MOS 6502 microprocessor is a great and important part of the entire software implementation. It will decide over the quality of the resulting handheld emulator. The following sections are going to workout the idea of emulating the entire MOS 6502 8 bit based microprocessor on an Atmel AVR microcontroller, which is also based on the 8 bit architecture.

One could imagine, that emulating an 8 bit microprocessor on an 8 bit microcontroller might be the most natural and easiest way because both rely on the same bus size. There must be taken plenty of control overhead into account because of the software based emulation of the 6502 hardware. Another difficulty is the fact that the microcontroller relies on a (modified) Harvard system architecture and the microprocessor on a standard Von Neumann system architecture.

3.2.1 Requirements and exclusions

There are several requirements, which need to be matched by the software emulation of the 6502 microprocessor.

- **Accuracy –** the software, which was written for the Apple][and its microprocessor, cannot and will not be touched by this thesis. It is the idea of an emula-

tor to imitate the behavior of the emulated hardware so that the original software will run without a notice. In order to archive this target, the operations of the 6502 microprocessor must be implemented with special care and accuracy to function like the original.

- **Speed –** the Apple][runs, as mentioned, at about 1,023,000 Hertz (1.023 MHz) [21] (p. 88). This amount of cycles is performed in one second by the original Apple][. The goal of this thesis is to create an emulation which is nearly 100% of the speed of the Apple][. Please note that a cycle exact execution cannot be achieved with the means used by this thesis, because it implements the whole emulator of the Apple][which includes also disk I/O, display I/O and keyboard I/O. They need to be handled as well and this will not allow to make a cycle exact MOS 6502 CPU emulation where every instruction takes exactly the same time as it would on the original. Empirical tests pointed out that slowing down or speeding up the clock slightly does not affect the operation[9].

  As described earlier, the 6502 takes between two and seven cycles per instruction, depending on the instruction and its addressing mode. Another premise is that the microcontroller will run at 20 MHz, so it is around a little bit less than 20 times faster. This leads to the simple mapping that 6502 instructions with two cycles have 40 cycles to execute on the microcontroller, three cycles have 60 cycles on the microcontroller and so on. This might sound like plenty of time, but we will see later on, that this in fact is too less time.

There are some specialities of the 6502 microprocessor, as described in the beginning of the thesis at section “Technical overview“ (p. 12) and “Instruction overview“ (p. 16), which cannot be implemented due to timing constraints and technical limitations:

- **the illegal instruction opcodes –** as previously conducted, the 6502 microprocessor has 151 valid opcodes out of 256 (one byte opcode length). So there remain 105 “unused” instruction opcodes. They are not wired to a defined nonsense operation (see “Issues on the 6502 and further decisions“, p. 18), instead they are all performing some action, sometimes stable and sometimes unstable. Application developers might have used these illegal opcodes, but for this thesis all opcodes which are not defined by the datasheet of the 6502 are treated as forbidden operations which will trigger an emulator exception.
- **implementation of some interrupts –** as covered in the section “Instruction overview“ (p. 16) there are three different kinds of interrupts: the NMI, IRQ, RESET and BRK. The NMI and IRQ interrupt are triggered by hardware. Since the emulation does not support external Apple][hardware, this interrupts are not implemented. The software IRQ interrupt, fired by the `BRK` instruction, and the RESET interrupt triggered directly by the RESET key on the keyboard will be implemented.

9 These tests were made with the AppleWin emulator (https://github.com/AppleWin/AppleWin), which allows the CPU clock frequency regulation. Varying the clock from 75% to 125% of the original clock speed does give only a minor change in the responsibility of the BASIC interpreter prompt or any other elements.

3.2.2 Designing the emulator – a simple approach

The target is now to design a C function to perform the actual 6502 microprocessor emulation. In order to do this, one should think of a really rough abstraction of one instruction execution control flow. It might be performed in this way:

1. reading opcode from memory
2. switching to the correct instruction opcode implementation
3. performing the actual execution or emulation of the instruction opcode; further memory accesses might be needed in this implementation
4. going back to step 1 and continuing with the next instruction opcode emulation

The simplicity of this roadmap makes it easy to create an initial piece of code, especially a simple function which covers all of these four steps. This function has the signature “`void exec()`”.

The length of the instruction opcode emulation is only a detail of definition: because of the fact that there is the need to emulate all other parts of the Apple][, e.g. the display or keyboard input, this function should only run at a fraction of the whole 1.023 MHz and then return to let the microcontroller perform other tasks. The function should emulate 51,150 clock cycles and then return. Twenty subsequent calls of this function will result in the desired 1,023,000 clock cycles. The emulator software can now watch for input from the keyboard and provide the display output between these emulation “blocks”.

Suspending the microprocessor emulation will bring up the need of saving the execution state of the entire microprocessor in order to continue at the same point when the `exec()` function is called next time. The state of the 6502 microprocessor consists of the following data:

- working registers: accumulator, x and y (all one byte in length)
- processor status register: p (one byte)
- program pointer: pc (two bytes to save)
- stack pointer: sp (one byte)

This core 6502 state will later on be extended by state meta-information about the emulator of the 6502 microprocessor itself and the graphics mode setting. But for now this state definition will work fine.

So these state fields will be implemented as global variables, which are stored in the SRAM of the microcontroller. With this basic idea the resulting code might look like:

```
1     unsigned char regA, regX, regY, regSP, regP;
      unsigned short regPC;

      void exec() {
5       unsigned short cycleCount = 51150;
        unsigned char tmp0;

        while (cycleCount) {
          // Fetch the next instruction opcode
10        tmp0 = memread(regPC++);
```

```
        // Switch to the right implementation
        switch (tmp0) {

15        // ...
          case 0x18: // CLC - CLear Carry
            regP &= 0xfe; // Clear carry flag
            cycleCount -= 2;
            break;
20        // ...

          default: // PANIC
            return;
        }
25    }
    }
```

Listing 1: a simple approach to the software emulator of the 6502 microprocessor.

Let us take a quick look at the source code: the outer while loop loops until all remaining clock cycles for the microprocessor are consumed. In every instruction implementation (e.g. line 18) the number of used clock cycles by this emulated instruction is subtracted. The function "`unsigned char memread(unsigned short addr)`" will read the memory value at the given address from the emulated Apple][main memory and return it. Because of the fact that this function is not so simple as one would think, it will be discussed later on in detail.

The given code shows for instance an implementation of the `CLC` instruction. Because of the simplicity of this instruction, there is only one important line of code: in line 17 the first bit of the processor status register will be cleared. All other instructions are implemented in the same manner.

But the key fact here is the point, that the function operates on global variables. On normal computers this might be a normal procedure to write code in this way, because it is the simplest way to access shared data from functions. This might not apply for our target platform, which is an embedded device – a microcontroller. So an excerpt from the compiled assembler code[10] looks like:

```
...
1     lds r24, regPC                ; 2 clock cycles
      lds r25, regPC + 1            ; 2 clock cycles
      mov r18, r24                  ; 1 clock cycle
      mov r19, r25                  ; 1 clock cycle
5     subi r18, -1                  ; 1 clock cycle
      sbci r19, -1                  ; 1 clock cycle
      sts regPC + 1, r19            ; 2 clock cycles
      sts regPC, r18                ; 2 clock cycles
      rcall memread
10    std Y+3, r24
      ldd r24, Y+3
      mov r24, r24
      ldi r25, 0
      cpi r24, 24
15    cpc r25, __zero_reg__
      breq .L6
      rjmp .L3
...
```

Listing 2: the compiled assembler code for the simple emulator approach.

10 Compiled / generated using: `avr-gcc 1_basic_emulation_function.c -S`.

Line one to nine represent the preparation of the parameter of the function call of "`memread()`" combined with the increment of the program counter variable. The C source code, from which this assembler instructions are generated, is a simple C statement: "`memread(regPC++)`".

Global variables, like the complete emulator state, are stored in the SRAM of the microcontroller, because the contents of these variables are preserved through function calls and can be changed from all locations in the program – possibly from an interrupt handler of the microcontroller. So the compiler cannot assume that the content of this global variable remains equal during the complete code execution of this function. So the compiler changes the code in this way that the data from the SRAM are loaded every time they are referenced and stored directly after a change. Then the value in the global variable is up-to-date at every time.

In the example from listing 2, the value of the regPC is loaded into the registers r25:r24 of the AVR. Because of the fact that the regPC is post incremented (after the function call of `memread()`), the values are copied to some temporary registers. On these registers (lines 5 and 6) the increment is performed. In this example using the subtract method. The incremented value is now on r19:r18 and gets directly written back to the SRAM. After this, the function `memread()` needs to be called. The registers r25:r24 still contain the program counter before the increment and this is the register pair, which is used by the `avr-gcc` compiler to pass parameters to the subroutine. So here is no further movement needed, the values are at the right place for the function call.

The summary of this little code snippet is that it takes 12 clock cycles for the microcontroller to simply perform an "`regPC++`" operation. Because of the fact that every single instruction emulation implementation will deal multiple times with multiple variables of the processor state, this assembler block will be placed in a same or equal way on every single usage.

Coming back to the `CLC` instruction from the top the emulator has 40 clock cycles on the AVR to execute the instruction and to be inside the speed requirement. So the next question is, how many clock cycles are needed to execute this instruction.

No	Description	(apprx.) Clock cycles used
1	Checking the condition of the while statement	6
2	Reading out the next instruction opcode and incrementing the program counter (12 cycles for `regPC++` and `memread()` call takes 4 cycles)	16 + time of `memread()`
3	Travelling trough the switch statement and finding the correct implementation	12
4	Execution of CLC	16
5	Jumping back to top and starting at the while condition proof	2 (1 if no cycles left)
	Total cycle count:	**53 + `memread()` time**

Table 7: rough calculation of the cycle count for a single instruction execution.

Table 7 provides a simple calculation of the cycle time for the example instruction `CLC`. Note that there is the need for much more than 40 cycles on the AVR. This emulator

would not run in realtime, because it is much slower than the original for this simple instruction. Furthermore there are other factors, which will make the code slower when it is fully populated with all instruction emulation implementations:

- the code and analysed assembler code consist only of one instruction. If all were implemented, the code would grow up, so that sometimes the limited range of branch instructions or the `rjump` will not be enough to get to the new code location. So the compiler is impelled to use instructions from the AVR RISC instruction set which will use more clock cycles and range longer and grow up the sum from table 7.
- the switch statement is not very complex with only one statement. So the compiler realized this in the example code as a simple if-condition. If there are 256 cases everything gets complex and there is the need for more cycles, which are used by the microcontroller to determine the right code section. Certainly the compiler will implement this quite smart (using a jump table or binary search) but it will take more cycles.
- the complexity or length of the `memread()` function. First of all the function call and return statement will take up to ten more cycles. Further this function needs to check many cases (as you know from the memory map section, p. 13) and will take a lot of clock cylces.

Putting all these thoughts together will lead to the conclusion, that this design is not as good as it needs to be to match the speed requirement.

3.2.3 Revision of the first approach

A very simple and common revision of the emulator software could be to remove the reference to the global variables, inside of the `exec()` function. As pointed out in the last section, this part consumes a great number of clock cycles. The sense of this effort is, that interrupt routines can disturb the execution of the `exec()` function and maybe change or read the global variables. So changes must be written back immediately to let them be actual at every time.

But for the purpose of this thesis there is no need, that the global variables are every time at the latest state, because only the `exec()` function reads and modifies this data. So with this thought the code can be modified to operate without making intense use of the SRAM.

```
1     unsigned char regA, regX, regY, regSP, regP;
      unsigned short regPC;

      void exec() {
5       unsigned char a = regA, x = regX, y = regY,
            sp = regSP, p = regP;
        unsigned short pc = regPC;

        /* original code using the local variables, see listing 1 */
10
        regA = a;
        regX = x;
        regY = y;
        regSP = sp;
15      regP = p;
```

```
    regPC = pc;
}
```

Listing 3: improved software emulator with local state variables.

This listing shows the usage of local variables. Beneath the SRAM, the AVR RISC microcontroller has 32 general purpose working registers, which can be accessed directly with the instructions. Placing the state variables in this registers will allow to speedup the emulation software, because the compiled code can directly operate on the data using the instructions. The state of the emulation is still captured using global variables, but only one at the beginning to fill up the registers and at the end to write back the changed data.

The new generated assembler code consists only at the top of `LDS` (Load Direct from SRAM) instructions and of `STS` (STore direct to SRAM) instructions at the end of the function. The body of the function is free from those SRAM memory accesses with exception to the `memread()` and `memwrite()` function. All the instructions are now executed directly on the register files of the microcontroller.

This way gives the best opportunity for optimizations, which can be achieved with the current code structure and from the high level language C. Optimizations inside the implementation of the instruction opcodes depend on the compiler and its optimization features.

3.2.4 Internals of the instruction opcode implementations

Until here only a general view on the emulation function `exec()` of the MOS 6502 microprocessor was given. Although this part of the code, which contains the main frame around all implementations of the instruction emulation, is an important part and we were able to optimize its execution speed. Another important part is the implementation of the instruction opcodes. Because of the fact that they are all constructed in a similar manner, the following section uses only one example for explanation. All other implementations only differ in minor differences and another semantic.

In order to take a look at the structure of the implementation of an instruction opcode, the example operation `ROL` in absolute addressing (`0x2e`) is used further. Omitting any details of the structure around, it looks like:

```
1   // Temporary variables
    unsigned char tmp0, tmp2;
    unsigned short tmp1;

5   // Get the global vars local
    unsigned char a = regA, x = regX, y = regY, sp = regSP, p = regP;
    unsigned short pc = regPC;

    // ...
10  case 0x2e: // ROL (abs)
      tmp1 = memread16(pc); // Address
      pc += 2; // PC increment by 2 (word addresss)
      tmp2 = memread(tmp1); // Read value
      tmp0 = (tmp2 << 1) + (p & 0x01); // Calculate Result
15    memwrite(tmp1, tmp0); // Write result

      // Adjust flags
```

```
        /* C */ if (tmp2 & 0x80) p |= 0x1; else p &= 0xfe;
        /* Z */ if (tmp0) p &= 0xfd; else p |= 0x02;
20      /* N */ if (tmp0 & 0x80) p |= 0x80; else p &= 0x7f;

        // Burn cycles
        cycleCount -= 6;
        break;
25    // ...
```

Listing 4: exemplary implementation of the ROL instruction opcode emulation.

The code listing starts with the definition of temporary variables (lines 2 – 3), which are used to store data during the execution of the instruction implementation. After this, the global variables are made local (lines 6 – 7), as worked out in the last section. This initialization is found only once at the beginning of the function. The actual implementation of the ROL instruction goes from line 10 to 24 and is located inside the while loop and switch statement from listing 1. To keep this example short, this code was omitted. The referenced functions inside the implementations are for emulating the Apple][main memory:

- `unsigned char memread(unsigned short addr)`

  As explained before this function reads the value of the emulated memory at the given location `addr`. As of the fact that the 6502 microprocessor has a 16 bit data bus the address is an unsigned short value.

- `unsigned short memread16(unsigned short addr)`

  Like the `memread()` function this function also reads a memory value. But it reads a 16 bit memory value from the location `addr` and `addr + 1`. The two values are handled in →little endian encoding and returned.

- `void memwrite(unsigned short addr, unsigned char data)`

  This function simply writes the given `data` at the given memory location `addr` into the emulated main memory.

These functions will be covered later on with a more detailed view. As it will turn out, these functions represent the last great barrier to reach the target of a speed accurate emulation of the 6502 microprocessor.

By this foreknowledge we are ready to analyse the actual implementation of the instruction opcode. The schemes of these implementations are easy. They consist of these parts:

1. **fetching of the instruction operand from memory** – lines 11 to 13. If the instruction works upon memory values, these are fetched at first. This step is not needed at every instruction implementation. Accumulator or implied instructions do not need any data from memory and work upon the internal processor registers.
2. **execution of the semantics** – line 14. The semantics of the instruction are executed. This part can be found on any implementation.
3. **write back of changed data to memory** – line 15. If the instruction works on memory values (e.g. like `ROL`), the computed result by step two is written back to a memory location. This step is also not part of every implementation.

4. **adjustment of processor flags** – lines 18 to 20. Some flags inside the processor status register are affected by this instruction. In this section the flags are set appropriate to the instruction semantics. This step is nearly always needed. On branch instructions, the flags are read to decide if the branch is taken or not.
5. **adjustment of the cycle count** – line 23. Every instruction lasts a firm amount of time on the original microprocessor. To create a sophisticated emulation, the emulator needs to track them. This step is present at every instruction implementation due to the fact that every instruction needs to "burn" cycles.

The first three steps are business as usual and can be performed in a easy and efficient manner due to the bit operators in C. The code for rotating left in line 14 is straightforward. The generated assembler code is also at its optimization limit.

More cycles of the emulator are consumed while setting the flags appropriate and adjusting the cycle count. These two parts are made in the original microprocessor by the hardware automagically without any need of further enhancements: the flags are output lines from the ALU with some logic gates and the cycles are not counted in the real microprocessor. They are consumed and nobody needs to keep track of them, because the instruction codes are designed to operate in this cycle count. But for the emulation these two steps are very important. These are the main locations where the overhead time and data of the emulation is generated.

As one can see in the actual example implementations, in lines 18 to 20, the implementation of this flag adjustment must be done using if–conditional statements. The outputs from the processor ALU are not available to the program.

With some tweaking of code and cross reading in the Atmel AVR datasheet and the `avr-gcc` documents one can find out that there is the possibility to read in the status register of the AVR microcontroller CPU which contains matching flags. This can be done by using inline assembler statements. Once the value is loaded to a temporary variable, the C code can use the flags set by the AVR microcontroller. The flags are set by the last assembler instruction executed. Due to code optimization of the compiler and the possible reordering of the C code no insurance can be given that the flags setting is the correct one for the performed operation in the last C code line. Besides, the checking of these flags would also result in if–statements as implemented in bare C code. So this optimization would bring no improvement to the code.

There might be some other ways like creating a "boolean" variable for every single processor flag and writing the values directly to it. All these ways will lead to other problems, which cannot be solved in a simple efficient way. The problem for these boolean variables is the reading, where further calculations might become necessary.

For example: if the carry flag is a boolean variable "`C`" and the code from line 20 is transformed into "`C = tmp2 & 0x80`", the execution at this point is fairly fast. If the carry flag needs to be added through the `ADC` instruction, there is the need for an if–condition to "normalize" its value to 0 or 1 and use it in `ADC` (since "`tmp2 & 0x80`" sets the eight bit and this results in a value of 128 or 0 for "`C`" and so 128 would be added through `ADC` instead of 1):

```
tmp4 = (signed int) (a + tmp0 + (C == 0 ? 0 : 1));
```

Another way is to shift "C", but without knowing which bit is set (other instructions might set the carry from the first bit and then the possible values are 0 or 1) there is a need of an further if statement (here the ternary operator).

If we add everything up, the emulation cannot get around this code and maintain the flags without if statements. So the optimization of this code cannot be done any further.

Since the structure of the 6502 emulation function was discussed, all instruction opcodes were implemented as proposed by the previous sections and resources of the 6502 microprocessor explaining the semantic of every instruction [17] [32].

3.2.5 Identifying other retardants

Up to here there is the question, if there is any part of the `exec()` function left which can be optimized any further.

And there is such a code block: the switch statement, which executes the right instruction opcode implementation. The MOS 6502 microprocessor has 151 different instructions which result in exactly the same amount of case statements inside the switch statement. The remaining 105 instruction codes are not used and mapped to the "default" case to catch an illegal opcode and halt the emulation. Because of the fact that the emulation is implemented in the high level language C, one has no more control over the mapping of the switch statement to assembler. The compiler has generally two options to optimize this code: generating a jump table or performing a binary search over the instruction opcode. Which one he chooses is determined for example by the distribution of the case value. In this case, the instruction opcodes are distributed over the whole 256 distinct possibilities (see table 3, p. 17) and the compiler might choose a binary search, because of the fact that the cases are not consecutive.

The selection of the compiler can be affected by some optimization compiler flags[11]. The compiler generated assembler code (default flags) takes around 27 clock cycles to reach the contents of the desired case-statement[12]. If we think of the time constraint from the beginning, this may lead to an important problem: for instructions of the 6502 microprocessor which take two clock cycles the AVR has 40 cycles to perform the emulation. If more than half of this amount is consumed by jumping to the right location, only around 13 cycles are left for performing memory access, computation of the result and adjusting the flags. As known from previous sections, there are other cycle consuming parts like the flag adjustment. This leads to the conclusion that this implementation has a speed problem achieving possibly only half the original speed whereas 6502 instructions with higher cycle count will probably not cause problems.

The problem at this point is that we cannot get rid of this speed problem from the high level C language. To find a faster solution, there are different other approaches to deal with the switch statement:

- **simulating jump tables in C [33] –** a static array is created with 256 fields and every field contains a reference to the appropriate label, e.g.

11 Assuming the use of `avr-gcc`: use "`-fno-jmp-tables`" to optimize switch statements.

12 Values acquired using the AVR simulator of AVR Atmel Studio 4.19-730 and the "`Os`" optimization level without further compiler flags.

“`static void *array[] = {&&foo, &&bar, ...};`“.

The jump to the label is then simply performed by the statement “`goto *array[i + opcode];`”. This solution takes up a great amount of memory inside the SRAM. If it is located in the program memory, it takes longer to load the data (Harvard architecture).

- **binary search with ifs** – theoretically a binary search needs only eight if–conditions to find to the suitable case block. Every if-condition is testing whether a bit is set and after a maximum of eight tests all bits are tested and the control flow of the program will be in the right place. In this scenario the code gets so long that the branch instructions and their limited jump range do not last to jump wide enough. A solution for that problem will slow down the execution.
- **inline assembler with C labels** – since version 4.5 of the `avr-gcc`, it is possible to jump from inline assembler to a C label. So implementing the time critical jump table in bare AVR assembler and jumping back to C labels for instruction emulation might do the job. Unfortunately this does not work very well due to some buggy behaviour of the inline assembler. Please note that this block would consist of jump initialization, 256 jump instructions and 256 inline assembler C label arguments. This makes the code confusing.
- **privilege of cases by splitting the switch statement into two switch statements** – as mentioned before, the compiler does not keep the order of the case statements. So arranging the instructions which are time critical at the top of the switch statement will have no effect. In order to favour these time critical statements, one can split the switch statement into two subsequent ones. The first contains all critical cases and the second one contains less critical to uncritical cases. Because of the fact that the first switch statement is closed, the compiler can resort inside this switch statement. The cases in this switch statement get checked at first and so they have a higher priority in contrast to the second switch statement.

These suggestions do not provide improvement at all with except to the last one. By splitting the switch statements it is possible to achieve a little performance improvement for the cases in the first switch statement. It now takes more time for the cases in the second statement to be executed, because there are two jump tables to pass. The key to reach the optimal optimization is the classification of the instructions for the first and second switch statement. Placing all MOS 6502 microprocessor instruction implementations with two or three clock cycles in the first switch and all other in the second one worked very well in the tests.

After this discussion of possible optimizations and the proposed optimization of the switch statement there is no further optimization possible with the current code.

3.2.6 The memory access

Until here the MOS 6502 microprocessor emulation was covered very deep in all its facets. Besides the bare instruction emulation there is a second, all-important key component: the memory emulation or emulation of the memory map. It will turn out

later on, that this is the main limitation in emulation speed. Albeit the memory map is not so much a part of the 6502 microprocessor emulation it will be handled together, because there is a very tight relationship between these two components.

The main purpose of this memory emulation is to implement the Apple][memory map (see table 4, p. 20) and the MOS 6502 microprocessor memory map (see section Memory map, p. 13). By nature, the memory emulation is quite simple: there is an array which represents the random access memory (RAM) of the emulated system. A memory access then is a simple array access. Because of the fact that the Apple][is based on the Von Neumann architecture there is no need for separate busses and memory for programs. Everything – data and programs – are located in the same memory (array).

This simplicity is broken by the memory constraints of the Apple][. The requirements of the 6502 microprocessor can be met very simple: zero page, stack and the interrup*t v*ectors are implemented by bare array accesses. But the requirements of the sophisticated Apple][memory map are a lot higher. There are different factors to consider (see table 4 for better understanding of the following topics, p. 20):

1. **the physical memory is limited (on a microcontroller)** – as seen later, it is not possible to emulate the full 48K of “user” memory of the Apple][. During a memory read process any address outside this range must be handled and the same applies to memory write.
2. **ROM contents are static & the microcontroller uses the Harvard architecture –** the static ROM with the Monitor system and the BASIC (upper 12K of the memory) does not need to be changed and can be stored as program data in the program memory section of the microcontroller. On read access of a memory address in that upper range (from `0xd000` on) the program memory needs to be accessed and the right values from program memory should be loaded. Write accesses to this locations should do nothing in order to not violate the memory integrity of the microcontroller (since this memory region does not exist in the SRAM).
3. **memory mapped I/O needs to be handled** – since the memory mapped I/O operations, like toggling the speaker to beep, were realized in hardware on the original Apple][. This must now be done by software and results in more overhead. On table 5 (p. 20) an overview over the different locations to handle is given. Due to restrictions of the microcontroller environment only a subset of this memory mapped I/O is implemented: keyboard, speaker and the graphics mode selection. If one of this locations is read there is a hardware action performed. Write actions should work also, because the 6502 instructions for writing will also read the location and perform the desired action.
4. **nothing performed in memory “gaps”** – there are some memory gaps, like the peripheral I/O memory, the extension memory and the missing memory inside the 48K “user” memory. All these location should return a constant value on read and do nothing on write to preserve the integrity of the microcontroller memory.

To put it into a nutshell, the main work of the memory emulation is done during mem-

ory read. During memory write there is only the constraint to write inside the emulator memory. As mentioned before, the following functions are responsible for this work:

1. `unsigned char memread(unsigned short addr)` and
   `unsigned short memread16(unsigned short addr)`

   to read from a memory location. The second function is only an alias for the first to read a 16 bit value in little endian encoding. This will not be covered deeper since it is only an alias function.

2. `void memwrite(unsigned short addr, unsigned char data)`

   simply to write contents to the emulated memory.

The source for the memory write function is straightforward:

```
1     void memwrite(unsigned short addr, unsigned char value) {
          if ((addr >> 12) < 3) // Check if inside array
              mem[addr] = value;
      }
```

Listing 5: source code of the memory write function in C language.

The data is only written into the array if the upper bound is not exceeded. Otherwise nothing happens as with real hardware, where no memory is attached: the data will be gone.

In order to read from the memory, different constraints must be checked and the code is more complex. The structure of the coded looks like (parts omitted):

```
1     unsigned char memread(unsigned short addr) {
        switch(addr >> 12) {
          case 0x0: case 0x1: case 0x2:
            return mem[addr]; // 12K RAM from the beginning
5
          case 0x3: /* … "case"s for 36K unused RAM … */ case 0xb:
            return 0;

          case 0xc: // Special I/O location
10          switch ((addr >> 4) & 0xff) {
              case 0x00:
                return keyLatch; // Keyboard read
              case 0x01:
                return (keyLatch = keyLatch - 128); // Keyboard reset
15            case 0x05:
                switch (addr & 0xf) { // Soft-Switches
                  case 0x0: // Graphics
                    return (grTxMode = 0);
                  case 0x1: // Text
20                  return (grTxMode = 1);
                  /* … more … */
                }

            }
25          return 0;

          case 0xd: case 0xe: case 0xf: // Monitor ROM & BASIC
            return pgm_read_byte(&(rom[addr - 0xd000]));
        }
30    }
```

Listing 6: shortened source code of the memory read function in C.

This source code for the memory read function is self-explaining: the constraints listed

above are matched with the switch statements. One could also use if-conditional statements. "`mem`" is the emulated memory array and "`rom`" contains the 12K ROM memory and is placed in the program memory section of the emulator[13]. The other variables are used for the keyboard key storage and the storage of the selected graphic mode.

3.2.7 First tests

Up to this point the emulation of the MOS 6502 microprocessor is complete in terms of running an assembler program. To emulate the Apple][computer system the display output, sound output and keyboard input is needed and will be added later on. But the running microprocessor and memory emulator can be used to do some first tests and take a closer look to the speed constraint.

As test environment the AVR microcontroller at the target clock speed of 20 MHz and an Arduino for speed measurement is used. (see "6.3 "Speed" measurement setup", p. 91). The program executed is the System Monitor program (input prompt, which wastes the time on checking the keyboard memory and waiting for input).

To the results: it takes about 146 – 160 ms to emulate 204,600 clock cycles of the 6502 microprocessor. This results in around 765 ms to emulate the entire 1,023,000 clock cycles of the 6502 microprocessor. Despite the number being under one second (time for 1.023 MHz of the original 6502) this time is bad, because the emulation consists of more tasks than shown up to here:

- **performing the screen output** – how this is going to work will be figured out some sections later, but this will be a time critical and consuming process, depending on the screen refresh rate.
- **loading pressed key codes** – as the original Apple][had a keyboard which performs all actions (key code lookup and storage of last pressed key) in hardware, the emulator must take care of this data and update it to reach a specific "key refresh rate". The Apple][also had key codes which are not compliant with modern keyboards. So there must be a translation performed. This depends on the keyboard used (also seen later on).
- **watching the state of the emulator** – if by example an unknown opcode is executed the emulator must handle this violation, because from now on the CPU is in an unsafe state maybe confounding data and program. So there is the need for conditionals which check these possibilities and watch the emulation.
- **distribution of emulation over the second** – the main time frame for the emulation is one second. In this time the emulator must run 1,023,000 cycles of the 6502 microcontroller. By simply emulating these cycles at a complete block and doing the other stuff in the remaining time, the input and output will become very poor. During this long blocking times no key is read or screen update is rendered. So the emulation must be divided into several parts, distributed over the second and the I/O tasks need to be performed in between. This is, because the `exec()` function runs 51,150 MOS 6502 microprocessor cycles

13 If "`avr/pgmspace.h`" is included and the target character array is annotated with "`const PROGMEM`", the access to this array – which resides now in the program section of the AVR – is done through the library function "`pgm_read_byte(...)`".

(the 20th part). So 20 function calls will work it and between these calls the other I/O operations can be done. This will create the illusion of basic “multitasking” and non lacking display output and display input. But this additional calls also take time not only for setting up the environment inside the `exec()` function (make global variables local etc.), but also performing the expensive subroutine calls.

All these facts lead to the conclusion that the created emulation is too slow to provide a speed accurate Apple][system. As discussed in the previous sections, all performance improvements are exhausted except for the final ultimate improvement: implementation of the MOS 6502 microprocessor emulation in assembler language as a last chance for human-based optimization.

3.2.8 Going back to the roots

Switching from the high level C code to assembler code is not as complicated as it may sound. The structure which was created during the last sections can be used further. On the other hand, the implementation in assembler will provide the complete control over optimization and realization. So the implemented assembler code will be as optimized as possible and extract the last bit of performance from the microcontroller.

The presented assembler implementation makes extensive use of assembler macros. Those macros are a bunch of assembler instructions which are grouped together and which can be parameterized allowing to replicate this code block multiple times inside the source code with different parameter registers.

In order to create working assembler code, some register conventions were declared and the registers were named through a macro to allow a later change and better usage. This convention is not discussed here, because it has no further importance; just some alias names for nicer code.

Memory emulation

Beside the fact that the source code now is in the assembler language, there is no great difference. The many if-conditionals in the memory read function may be implemented a bit more efficient. A further speed improvement comes from the fact that the memory read function is now realized as an assembler macro. Every occurrence is replaced by the instructions grouped by the macro, making the subroutine call and return needless and saving time with this advance.

The drawback of this “improvement” is the code size: the memory read macro has a length of approximately 101 lines (without blank lines and labels) and every line stands for an instruction. On every place in the emulator implementation, where the memory read macro is embedded, these 101 instructions are replaced. And because of the fact that this macro is used extensively, the code gets really large. But that problem is, as an exception, not important for further considerations.

Because of the length of the memory read macro, only an excerpt is shown to demonstrate the structure:

```
.macro memread ah al
cpi \ah, 0xd0
brlo 2f
; -- EXCEUTED, IF ADDR > 0xcfff --
  mov ADDRTMP, \ah
  subi ADDRTMP, 0xd0
  movw ZL, ROML            ; ROMH:ROML = array base address[14]
  add ZL, \al
  adc ZH, ADDRTMP
  lpm RES, Z               ; Load data
  rjmp 4f                  ; Finished
2:
cpi \ah, 0x30
brsh 3f
; -- EXCEUTED, IF ADDR < 0x3000 --
  ldi ZL, lo8(mem)
  ldi ZH, hi8(mem)
  add ZL, \al
  adc ZH, \ah
  ld RES, Z
  rjmp 4f                  ; Finished
3:
clr RES                    ; Nothing to load
cpi \ah, 0xc0

breq 26f                   ; -----------
rjmp 4f                    ; Extend jump
26:                        ; -----------

; --- Handle access in 0xc0xx ---
  ; ...
  7:
  cpi \al, 0x40
  brsh 8f
    ldi ADDRH, 0x01        ; 0xc030 - 0xc03f: Speaker toggle
    in ADDRTMP, 0x05       ; Speaker is connected on PB0
    eor ADDRTMP, ADDRH     ; XOR for toggle
    out 0x05, ADDRTMP      ; Write back port data
    ldi ADDRH, 0xc0        ; Reset high address byte
    rjmp 4f
  ; ...
  ; ------------------------------
; ...
4:                         ; Finished
.endm
```

Listing 7: condensed assembler macro to read from emulated memory.

The macro takes two arguments (line 1), which are replaced by the registers, which contain the address word, locating a value in the memory. The result is later stored into the register "`RES`" (alias for r21). The macro contains at its end a variable label called "`4`". Whenever a memory read operation has been finished the control flow jumps to this location, omitting all other branches to save time. Because of the extraordinary length of the macro sometimes the branch instruction does not stretch wide enough to reach the label "`4`". So there is the need for jump "extension" code, e.g. lines 26 to 28.

If the comparison in line 24 is not true the branch in line 26 needs to be "`bne 4f`" – if not true then go to the end of the macro. Due to limited range of the branch instruc-

14 In order to "load" or work with different ROMs containing the BASIC language the base address of the array in program memory, containing the ROM data, is placed at the beginning of the emulation into a register pair (16 bit value) and then loaded from there. See section "The main frame of the emulation function" (p. 49) for further details.

tion, the opposite test needs to be used which will lead to a "`rjmp 4f`" if they are not equal and skip this relative jump if they are equal and continue execution in line 32.

The ROM address is checked at first and followed by the RAM address. It is most likely that the next byte will be read from ROM because of the fact that the "operating system" (BASIC and Monitor program) is located there. But it might also be possible that the program wants to access some data, e.g. BASIC program lines in the RAM. In contrast to this it occurs most seldom that the memory mapped I/O is accessed. So this is located at the end of the macro.

One can see that a ROM and RAM address take "only" around 13 AVR clock cycles and a memory mapped I/O address lasts longer depending on the location. This is a quite good result, because the solution from section "The memory access" (p. 40) with dedicated functions takes at least 10 AVR clock cycles[15] to get only *into* the function. So it will take a lot more AVR clock cycles then this solution.

From line 35 to 40 the memory mapped I/O code for the sound output is shown. The code simply flips the logical level on the output pin of the microcontroller, creating a short noise. Steady repetition will create a tone. If a memory read is executed at this address (`0xc030` to `0xc03f`) it will take around 18 AVR clock cycles.The other memory mapped I/O implementations differ only in their function, but share the same structure. They are omitted here.

The macro for writing into memory is quite simple:

```
1       .macro __memwrite__
        mov ADDRTMP, ADDRH         ; Check if we are exceeding our memory
        cpi ADDRTMP, 0x30          ;  limit of 0x3000
        brsh 1f
5
        ldi ZL, lo8(mem)           ; Load memory array offset
        ldi ZH, hi8(mem)
        add ZL, ADDRL
        adc ZH, ADDRH
10      st Z, RES                  ; Write data

        1:                         ; Continue if out of memory
        .endm
```

Listing 8: complete memory write assembler macro.

It checks – like the C variant – if the desired address is inside the memory array and writes the data only then. This macro has no arguments; the value to write is stored in "`RES`" and the address to write to is stored in the register pair "`ADDRH:ADDRL`". It takes around 3 AVR cycles if the address is out of bounds and around 10 AVR cycles if the address is in range. In comparison to the C function this macro saves a lot of cycles.

Memory emulation and addressing modes

One point, which has not been covered until now, is the implementation of the addressing modes. In the C MOS 6502 microprocessor emulator code the different addressing modes – to read or write a memory value – are implemented directly into the instruction implementation. For example reading a value with the indirect indexed

15 Assuming, that instructions "`CALL`" (5 cycles) and "`RET`" (5 cycles) are used due to limited range of relative calls. Please note, that beneath the subroutine call and return also the used registers are saved onto the stack and restored. This takes cycles too.

(*indY*; see table 2, p. 16) addressing mode looks like:

```
memread(memread16(memread(pc++)) + y)
```

To keep the code nice and organized macros are used for these addressing modes, too. The structure of these macros is very simple: they use the program counter of the emulated CPU as well as the address registers to load the value onto "`RES`". So after the macro, "`RES`" contains the memory value and "`ADDRH:ADDRL`" the intended address . The program counter is also incremented while the instruction bytes are read to get the memory address. The example of the indirect indexed addressing mode as assembler macro looks like:

```
1     ; Read the next byte of the program counter and increment
      .macro __memread_pc__
      memread regPCH regPCL
      adiw regPCL, 1
5     .endm

      .macro __memread_indY__
      __memread_pc__            ; Get the zero page address from arg
      clr ADDRH
10    mov ADDRL, RES

      __memread__               ; Read the 16 bit word at the zero page
      mov TMP0, RES
      ldi TMP1, 1
15    add ADDRL, TMP1           ; Next memory location contains ADDRL
      adc ADDRH, ZR
      __memread__
      mov ADDRH, RES
      mov ADDRL, TMP0
20
      __cyc_save_cross_addr__   ; Save original address

      add ADDRL, regY           ; Add value of Y register
      adc ADDRH, ZR
25
      __memread__               ; Read result value
      .endm
```

Listing 9: addressing macro to read a indirect indexed value.

This takes many AVR clock cycles, due to the fact that the memory is read four times. But one should consider that this addressing mode is very sophisticated. Instructions, which use this addressing mode, take around six to seven 6502 clock cycles. That means they can take between 120 and 140 AVR clock cycles. Additionally, the cycle count of instructions with this addressing mode is variable: if a page border is crossed, one more 6502 CPU cycle is consumed. The macro in line 21 manages a part of this, but it is not expounded closer. The other addressing modes are implemented in the same manner.

The jump table

As worked out before, the switch statement in C is a main component. It is the target to find a solution which performs the "way" to the requested instruction opcode emulation implementation as fast as possible. The solution used here is a static jump table with constant cycle count to pass.

```
1     __memread_pc__             ; Get next 6502 opcode; RES = opcode

      ldi ZL, pm_lo8(jmptable)   ; Load memory offset of jump table
      ldi ZH, pm_hi8(jmptable)
5     add ZL, RES                ; Add argument twice, because jmp is
      adc ZH, ZR                 ;  32 bit wide
      add ZL, RES
      adc ZH, ZR
      ijmp                       ; Indirect jump to location
10
      jmptable:                  ; The jump table
      jmp brk_0x00
      jmp ora_0x01
      jmp dummy
15    jmp dummy
      jmp dummy
      ...
```

Listing 10: the structure of the used jump table.

This jump table has a constant cycle count of eleven (11) AVR cycles per pass. In comparison to the number of AVR cycles needed by the switch statement (see “Identifying other retardants“, p. 39), this assembler variant is more than 50 percent faster.

The principle is simple: in line 1 the next opcode is loaded into “RES” (does not count into cycle count of eleven). Once “RES” (the target emulation instruction) is known, the location of the jump statement in the jump table, beginning at line 11, needs to be calculated. In order to do that, the “base” address of the jump table label is loaded into the Z register and “RES” is added twice. The result now points to a jump instruction which brings the program to the right case.

An AVR “jmp” instruction has a 32 bit opcode and takes two →words, twice the program architecture size (Harvard architecture). So the first jump instruction is located 0 words after the base of the jump table, the second two words after, the third four words after and so on. Every valid jump instruction of the jump table is a multiple of two (words). Not adding twice the target index in the jump table would sometimes point to the second word of the jump instructions, since they are two words long. This would cause wrong behavior of the microcontroller.

Once the new location is computed inside the Z register, the indirect jump instruction is used to jump to the specific location in the jump table and then it jumps from there to the right label with the opcode implementation. The jump table has 256 entries. All illegal opcodes are mapped to a “dummy” label which will trigger the error of an unknown opcode and exit the emulation.

Instruction emulation in assembler

The implementation of the instructions in assembler does not differ very much from the structure of the C implementations (see “3.2.4 Internals of the instruction opcode implementations“, p. 36). First the arguments are read, then the computation of the result is performed, the result is stored, the flags are set and the cycle count is subtracted from the remaining 6502 cycles. The following example in listing 11 gives an insight into the look and feel of these implementations.

The structure follows the C structure. But it needs to be pointed out that the setting of the flags can be implemented more sophisticated and more efficiently by using less

cycles. Also the AVR assembler instructions are used to perform the action intended by the opcode:

```
1     ; ROtate Left (abs, 6)
      rol_0x2e:
      __memread_abs__
      bst RES, 7                          ; Save the shifted out
carry bit
5     sec                                 ; Transfer the 6502 carry
bit into
      sbrs regP, 0                        ;  the AVR status register
      clc
      bld regP, 0                         ; Set the new carry bit
      rol RES                             ; Rotate left with AVR
carry
10    __memwrite__                        ; Writeback result
      __nz_flags__ RES
      __cyc__ 6
      jmp loop
```

Listing 11: implementation of ROL instruction in assembler.

In this example, the AVR “`rol`” instruction is used directly in line 9. In line 5 to 7 the AVR carry flag is adjusted to the one of the 6502 processor status register to perform the computation.

Also the T flag, which is a general purpose flag on the AVR and its related instructions “`bst`” and “`bld`” are very useful while transporting the bits from the AVR status register to the emulated 6502 status register and vice-versa.

After computing the result (line 9), it is saved back to memory (line 10). Due to the fact that the memory read macro (line 3) saves the address of the read location to “`ADDRH:ADDRL`”, the memory write macro can reuse it without recomputing it. After that the remaining flags are set (line 11). In this case the N and Z flag of the 6502 microprocessor. Because of the fact that these two flags are often set together, they are wrapped into their own macro. Then the cycle count of this instruction is subtracted (line 12) and the emulation of this instruction is finished with jumping back to the top (line 13) and fetching the next opcode.

This is the very simple structure, which is followed by all instruction emulation implementations. It needs to be pointed out that implementing it directly in assembler gives more control over the speed, but also improves the accuracy of the emulated instruction. By comparing the MOS 6502 microprocessor →ISA [17] [32] and the AVR RISC ISA [34] for a single instruction one can simply work out the differences and map 6502 instructions to AVR instructions. There is no need to reimplement them. With this one can save work and benefit from the performance of the instructions. To a specific degree, this is only a mapping of 6502 instructions to AVR instructions. It helps to ensure the correctness of the implementation.

The main frame of the emulation function

The jump table, instruction emulation and memory access macros form the main core of the emulation function. They are embedded into the “main frame” which is responsible for counting the clock cycles, loading global variables into registers and exiting the emulation function. The abbreviated code looks like:

```
1    .global mos6502_exec
     mos6502_exec:
       __init__                     ; Save regs and load global data

5      lds TMP1, is_apple2_plus     ; Load the address of sel. ROM
       cpi TMP1, 0x00
       brne ldrom2
       ldi TMP1, lo8(apple2ROM)     ; If use_rom1 == 0 : use ROM1
       mov ROML, TMP1
10     ldi TMP1, hi8(apple2ROM)
       mov ROMH, TMP1
       rjmp skiprom2
     ldrom2:
       ldi TMP1, lo8(apple2PlusROM)   ; If use_rom1 != 0 : use ROM2
15     mov ROML, TMP1
       ldi TMP1, hi8(apple2PlusROM)
       mov ROMH, TMP1
     skiprom2:

20     lds CL, unused_cycles        ; Cycle counter = 51150 cycles
       clr CH                       ;  + cycle offset from r24
       ldi TMP1, 0xce
       add CL, TMP1
       ldi TMP1, 0xc7
25     adc CH, TMP1
       rjmp loop                    ; Skip 6502 pc increment

     loopPCInc:
       adiw regPCL, 1
30   loop:                          ; Begin of main loop
       cpi CH, 0                    ; Check: is cycle counter 7 or
       brne nonstop                 ;  lower -> not enough cycles
       mov TMP1, CL                 ;  for the next instruction
       subi TMP1, 7
35     brpl nonstop
       jmp end                      ; Return
     nonstop:                       ; Go on silently

     ; ... ... ... the jump table with implementations ... ... ...
40
     end:                           ; Normal exit
       sts unused_cycles, CL        ; Unused cycles
       sts emulator_state, ZR       ; Everything fine
       __final__
45     ret
     dummy:                         ; Error exit: illegal opcode
       sts unused_cycles, CL        ; Unused cycles
       ldi RES, 0xfe                ; Error on illegal opcde
       sts emulator_state, RES
50     __final__
       ret

     ; ... ... ... other exit labels ... ... ...
```

Listing 12: condensed main frame of the 6502 emulation function in assembler.

The main frame is segmented into different parts. At the very beginning, the registers used by this function are saved onto the stack and the global variables are loaded into the specific registers. This is done by the “`__init__`” macro (line 3). Because of the fact that it is really simple in its structure (only SRAM load and stack push instructions) it is not covered any further here. Before leaving the function the register values needs to be saved to the global variables and the saved registers should be restored. This is done by the “`__final__`” macro before a “`ret`” (e.g. line 45 or 49). The used AVR regis-

ters must be saved and restored, because they are replaced by other values, but needed by the calling program.

After setting up the registers the function is ready to start. There is the need to make further arrangements: from line 5 to 18 the base address of the ROM is loaded into "ROMH:ROML". This is necessary to allow the emulator to run with different ROMs. For example the original Apple][had the Integer BASIC ROM whereas the Apple][plus came with the Applesoft BASIC ROM. Loading the base address of the used array into a variable in dependence of the state of the global variable "is_apple2_plus" allows the emulator to switch between the different ROMs reclined in the AVR program memory. The code is simple: it is an if–conditional statement to load either the one or the other array base address into the register pair.

The next task for preparation is to setup the cycle counter. Once this function needs to perform 51,150 MOS 6502 microprocessor cycles, the counter needs to be an unsigned short with a length of 16 bits. The registers "CH:CL" are reserved exclusively for the counter. The cycle counter counts down using the "subiw" instruction. From line 20 to 26 the counter is loaded with the static value. But there also is the value of the global variable "unused_cycles" added (line 20). This global value stores the unused and not consumed cycles from the last emulation execution.

Because of the fact that the 6502 instructions have a varying cycle count – between two and seven cycles – not all 51,150 can be consumed. For example: if there remain less than seven cycles and a seven cycle instruction is executed, the emulator will use more than the remaining cycles and the cycle counter register becomes negative. The binary number wraps around and because it is assumed that the value is unsigned, it will continue executing (since it is a very great number now) the emulation and will not return. To prevent this effect, the function returns if there are less than eight cycles and stores the remaining amount in this global variable to be used next time. At the beginning of the main loop (line 30) the cycle count is checked at first. This is another simple code block performing two conditional statements: if the condition is true, the program jumps to the end label. Otherwise the execution continues with line 39. This line contains the code from listing 10 – the jump table and instruction implementations. The "loopPCInc" label (line 28) is simply used to increment the program counter before executing the next loop.

At the end of the function (line 40 and following) the registers must be saved and restored as described through "__final__". Before doing that the global variables "unused_cycles" and " emulator_state" are set (e.g. line 42 and 43).

The global variable "unused_cycles" has the meaning as described prior. The global variable "emulator_state" stores the state of the emulator. There are multiple "end labels" which differ in the value set to this variable. For example if the program jumps to label "end" this variable is cleared, because no error happened. But if the program jumps to label "dummy" it is loaded with the value 0xfe, indicating that an illegal opcode was used.

The calling "program" of the exec function can determine, after the return of the emulation function, by reading this variable, if any error happened and do some error handling actions. There are more such "end labels" sharing the same structure, catching

errors or events like usage of decimal or high resolution graphics mode.

Details of development and testing

The presented parts form the assembler code implementation of the MOS 6502 microprocessor emulation. But there is a last outstanding question: "how was it developed and tested?".

Because of the fact that the assembler code is specific to the Atmel AVR RISC architecture it cannot be executed on x86 computers directly. So for the development of this specific module AVR Studio together with the AVR Simulator was used.

The implementation of the 6502 instructions should not contain any faults, because of the fact that the emulation relies on the AVR instructions. To ensure the highest code accuracy every instruction in every addressing mode was manually tested during development using the AVR Simulator of AVR Studio.

3.2.9 New speed measurements & summary

After taking a closer look at the assembler implementation and putting it together to a runnable version, it is time to check the performance of this new implementation. As described in section "3.2.7 First tests" (p. 43) the C implementation takes about 146 – 160 ms to emulate 204.600 clock cycles of the 6502 microprocessor.

Using the same test, with the same test conditions and replacing the C implementation with the created assembler implementation results in 118 – 120 ms to emulate those 204.600 MOS 6502 clock cycles. This is a speedup of around 22% in contrast to the C implementation. So it will take around 600 ms to emulate the 1.023 MHz of the MOS 6502 microprocessor.

This sounds very good and might give the other operations enough time to perform in one second. The final speed measurement will be done at the end – in "5.1.1 Achieved emulator speed" (p. 82) –, when all components are assembled and running to get some realistic performance data. Be aware of the fact that the test was done with the Monitor program, but there was no real load – only the input prompt.

Finally, this subchapter shows the evolution of code in order to match the constraints of speed accuracy and accurate implementation and describes the successful way resulting in a highly optimized assembler code to emulate the MOS 6502 microprocessor and the memory map of the Apple][system. Please note that the assembler code is so highly optimized that it might not be possible to get any greater performance improvements out of the emulator.

3.3 The emulator runtime environment

Up to now the MOS 6502 microprocessor emulation module was created. Although this component is the main part of the emulator, there are other important parts which will lead to a usable handheld device. For example some sort of display output or keyboard input are needed in order to use this emulator. In the following those parts are called the “emulator runtime environment”.

Beneath these tasks the runtime environment has some further tasks: it needs a possibility to exit the emulation and perform management operations like resetting the emulator or loading programs into the memory of the emulator. This special features will make the result more useable, for example if there is an option to load original software and run it on the emulator. The variety of this features are limited by some factors:

- **program code size** – it might not seem so, but the code so far has a size of around 98K (100,826 byte) and borrows around 77% of the program memory. The 98K can be divided into 12K for each of the both ROM memory arrays and 75K for the CPU emulation. The CPU emulation consumes this great amount of size, because of the fact that the really heavy memory read macro is replicated a lot more than 151 times (at least one memory read usage on every instruction implementation), giving a lot more than 15,251 lines of assembler code (the memory read macro has a length of 101 lines without blanks and labels). So only 23% of program memory is left for the code size of the runtime environment.
- **SRAM size** – by constraint the implemented Apple][system has 12K of main memory. So 75% of the static RAM of the AVR microcontroller are used by the Apple][memory. Due to the fact that AVR programs also need space in the static RAM for variables the runtime environment is limited here.
- **time** – having nice features is great, but also the time for this thesis is limited and nice features take a lot of time to be developed. So the remaining time for the project will decide which features will get implemented.
- **systemic restrictions** – due to the fact that the entire Apple][is emulated on a single AVR microcontroller there might not be enough resources for some features or they are not possible, because they violate system restrictions. By example the Apple][system had the possibility to save and load binary data to and from a standard cassette tape. This cannot be implemented, because it has hard realtime requirements (bit duration etc.). But the emulator will not allow this precise timing and also the divided structure of the `exec()` function in twenty calls prohibit some bit detection system. And due to the fact that the AVR is not aware of multitasking, this feature cannot be implemented with the current structure.

With these limitations and the target of an usable, single microcontroller based, portable Apple][emulator the following features will get implemented further:

1. **backend menu** – this menu serves as graphical user interface to the emulator runtime environment and allows the user to perform one of the following

actions. It should be a simple text based backend menu using the keyboard to make selections.

2. **hibernation** – the entire emulator state can be saved to one of ten hibernation slots. It will be stored in a non-volatile storage, like an EEPROM. There should be the menu option to load this state back into the emulator and continue the emulation later on.
3. **loading of disk images** – it would be tedious to deal only with the BASIC prompt. So loading programs from a floppy image[16] would give it more attraction to try old software. At this point there is the need for a tradeoff: due to speed limitations of the emulator along with hardware and time restrictions, the Disk][[17] will not be implemented. Instead there will be a textual assistant in the backend menu, which can load a program from a disk image directly into the emulator memory. The disk image is located on an SD card.
4. **resetting the emulator** – it should be possible to simply reset the entire emulator to the state of a fresh Apple][.
5. **adjustment of backlight** – as the target is a handheld device it will have a display. Displays need to have a backlight to display content. The brighter the display backlight is, the more current is consumed and the battery will last shorter. So a simple adjustment option will give the users the possibility to adjust the backlight to their wishes and extend the battery life. In the backend menu there should be a slider to let the user adjust the backlight setting. Because of the fact that this is done by hardware it will be covered later on.

3.3.1 The structure

In figure 6, an overview over the overall code modules forming the entire emulator developed in this thesis, is given.

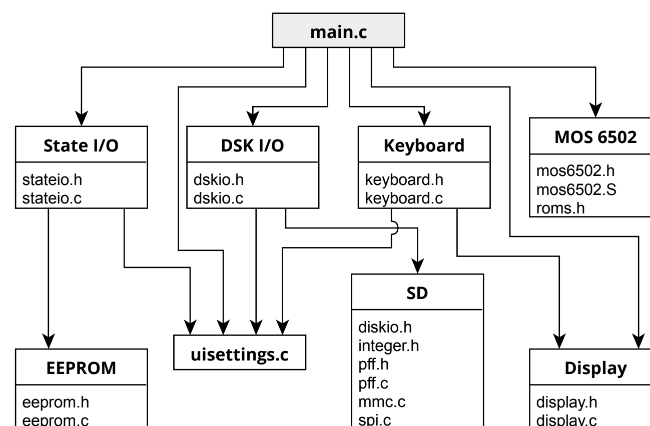


Figure 6: module and file structure of the Apple][emulator project. (The arrows express a usage relation.)

16 Please note that it might be illegal to download disk images from the internet without having them originally or having a permission of the publisher.

17 The “Disk][“ was a floppy drive for the Apple][, which was developed slightly after the release of the Apple][original [35] [36]. It came with an extension card and was plugged into an extension port on the Apple][main board. Then the driver software was displayed in the main memory (in the region of page `0xc1` to `0xc7`) due to memory mapped I/O. As known from the appropriate sections the memory read function is critical regarding speed and code size.

As visible, the entire emulator consists of nine modules with a specific dedicated task to every module. Upon here the module “MOS 6502” was discussed deeply. The other modules together form the runtime environment of the emulator, maintaining all kinds of utility work and I/O management. Together with the Makefile these files form the entire software needed for the Apple][emulator.

The “SD” module manages the access to a MMC / SD storage card with FAT16 or FAT32 file system. It is further used to load program data from disk images into the emulator. The disk images can be placed on any microSD card on a computer or other device and then simply read in by the emulator. Due to the fact that developing a working library to read SD cards is its own dedicated topic for a thesis, in this case a working library was used. Here the popular library “*Petit FAT File System Modul*” from *Elm Chan* is used [37]. So the files or internals are not described here, but rather only used.

The module “uisettings.c” only contains some C definitions for the user interface, like font and background colors and other utility information. This allows later on a simple change of the →UI appearance if there is the need for it.

The module “main.c” is the main file, containing the main function as the entry point of code execution. The main emulation loop and backend menu of the emulator runtime environment are implemented here.

3.3.2 Display output of the emulator (module “Display”)

The video output is one important feature for the emulator itself and the runtime environment to display the backend menu. There are many options for video output, such as generating Composite video or VGA output from the AVR emulator. Libraries are existing for all those options. But this video output modes are realtime driven. That means that the display is painted line by line very often during a second. This leads to a very high utilization of the microcontroller only with refreshing the screen. Around 90% of the time is spent on video output. But as we kno, the emulation takes the main part of time of the CPU. So this is not an option. These external devices also do not lead to a handheld device.

So in this thesis, a LCD display module with GRAM (graphical RAM) and display controller will be used. These display modules are connected through a bus and control lines to the microcontroller and can exchange data over the bus. The main point is that the graphical RAM stores the pixel information and the microcontroller can decide when he updates the display data. Until this event the pixel data are shown as set in the GDRAM the entire time without the need of refreshing them. This gives the software the freedom to run the emulation. The details over connection and interfacing the display and its type will be given later on in section “4.1.2 Interfacing the display” (p. 66).

The “Display” module from figure 6 provides some public interface functions to let the main display drawing function, located in main.c, draw the display.

```
1   void lcd_init(); // Setting up the display
    void lcd_clear(unsigned short color); // Clearing the display
    void lcd_apple2_text(char x, char y, unsigned char c, char
        flashHigh);
5   void lcd_apple2_lores(char x, char y, unsigned char block);
```

```
void lcd_printa2c(char x, char y, char c);
void lcd_printfp(unsigned char x, unsigned char y, const char
      *string, ...);
```

Listing 13: public interface of display.c.

Listing 13 lists all interface functions provided by the display driver. The first two functions are needed during initialization to connect to the display and to clear the entire screen with a given color.

The functions in line four to six are used to display the Apple][screen. They provide the basic function needed by the text mode and low resolution graphics mode. As intended by the function names, “`lcd_apple2_text`” draws an Apple][character (see section “Text mode“, p. 22) onto the screen at the specified location and “`lcd_apple2_lores`” draws two low resolution blocks with its colors (see section “LoRes graphics mode“, p. 23) onto the screen. Using this functions it is possible to create the function “`draw_apple2_display`” (in main.c) to render the screen. It iterates through the screen memory and uses one of the prior functions to bring the characters or graphic blocks onto the display.

The last function (line 8) is used for the emulator runtime environment backend menu, to have a text based console output. It uses a bigger font (8x16 pixels vs. 7x8 pixels of the Apple][font) to display a more readable menu.

This very simple interface allows a powerful display output. The display draw function needs only be called between the `exec()` function calls for 6502 microprocessor emulation to render the display output of the Apple][.

One further note: the function “`lcd_printfp`” is a simple reimplementation of the famous “`printf`” string output function. Because of the fact that every “normal” string eats up static RAM this function operates directly on strings, which are placed in the program memory consuming only this type of memory. The macros “`lcd_printf`” and “`lcd_print`” in display.h perform this action automatically and hide this behavior to the programmer.

3.3.3 Keyboard input (module “Keyboard”)

The next important part to make the emulator usable is the keyboard for data input. To create a portable handheld as result the keyboard was build from scratch. More details are discussed later on (see “4.2 The “keyboard” microcontroller”, p. 72).

As seen many times before, CPU time is very rare. And building a keyboard from scratch brings the need of a keyboard matrix with a permanent scanning through this matrix to detect any key presses immediately. Because of the immediate scanning process this cannot be done on the emulation microcontroller. For that reason the keyboard will be sourced out to a second smaller microcontroller, referred as “keyboard controller”.

Another reason is that the key code of the Apple][keyboard differs from modern keyboards just as the Apple][keyboard contained some special keys which are not existing any longer. So there is the need for a mapping of key codes. This means further CPU utilization, even if it is not big. But also keys like the REPT key (see “The keyboard“,

p. 24), which replicate a steady key press, will result in more CPU utilization. And a keyboard from scratch might be smaller to fit into the case for the handheld device.

The task of the keyboard controller is to scan for key presses, translate the pressed key into an Apple][key code and transmit it via serial communication to the main emulation microcontroller. This advance brings two benefits with it:

- **serial communication implemented in hardware on the microcontroller** – this means that the emulation microcontroller can receive a key code without executing code for it and spending time on waiting for responses. The emulation microcontroller only needs to check if a receive bit is set and can read the received key code directly from the serial buffer. This causes minimal CPU time consumption.
- **abstraction through an interface** – the two controllers are communicating through the serial communication. This allows later on the replacement of the DIY keyboard with a a PS2 or USB[18] keyboard without changing the emulation microcontroller. Only the keyboard microcontroller must be updated with a new firmware capable of the new device. This abstraction makes the design more flexible for further development.

The “keyboard” module provides these public functions:

```
1    void keybrd_init();
     char keybrd_get_ascii();
     short keybrd_get_num(unsigned char x, unsigned char y);
     unsigned char keybrd_prompt(unsigned char x, unsigned char y,
5           unsigned char length, char *dest, unsigned char offset);
```

Listing 14: public interface function of keyboard.c.

The first function is used to setup the serial communication interface and should be called very early in the main function.

The function “`keybrd_get_ascii`” is the main workhorse, waiting for a key code from the keyboard controller. Because of the fact that the keyboard sends Apple][key codes, they are converted to an corresponding ASCII character. Please note that this function blocks until a character is available in the serial communication buffer. The last two functions (lines 3 to 5) are some helper functions to provide an input prompt for numbers or strings as UI (user interface) elements in the backend menu.

The entire library is only used by the backend menu of the emulation runtime environment to provide some basic user interface input for the users to let them control the menus.

The keyboard input during the emulation is done directly in the memory read macro in assembler (see “Memory emulation“, p. 44) to achieve the maximum speed. The intended part of the memory read macro is:

```
1    ; ... ... ... other parts of macro memread ... ... ...
     ; --- Handle access in 0xc0xx ---
     cpi \al, 0x10
     brsh 5f                    ; Keyboard character read
5      lds RES, UCSR0A          ; Check if data received via UART
```

18 Interfacing a PS2 keyboard with a microcontroller is fairly easy and there are many libraries performing this task. Interfacing USB devices is more complex, but still possible. HID (human interface devices) can be interfaced through the USB 1.0/1.1 low speed protocol and can be interfaced by a microcontroller. See http://www.asahi-net.or.jp/~qx5k-iskw/robot/usbhost.html.

```
        sbrs RES, RXC0
        rjmp 16f              ; If not: jump to keyboard end label

        lds RES, UDR0         ; Load keyboard value
10
        cpi RES, 0xf0         ; Check if special action needs to be
        brlo 17f              ; performed (MENU / RESET)
        jmp resend

15      17:
        sts key_latch, RES    ; Store new value
        rjmp 4f               ; Finished

        16:
20      lds RES, key_latch    ; 0xc000 - 0xc00f: Keyboard access
        rjmp 4f
      5:
      ; ... ... ... other parts of macro memread ... ... ...
```

Listing 15: condensed assembler code of receiving characters via serial communication.

In line five it is checked if the microcontroller received a data byte through serial communication. If this is false, he jumps directly to line 20 and reads out the value from the key latch, storing the value of the (last) key code (see section “The keyboard“, p. 24).

If there is serial data in the one byte buffer, it gets read out on line 9 to the register “`RES`” and stored on line 16 to the “`key_latch`” variable. Because of the fact that “`RES`” is the register with the “return” value of this macro the key value can remain there.

Since there are some special keys on the keyboard there is the need for further checks. Whenever the “SHIFT + RESET” or “CTRL + RESET” (to get into the emulation runtime environment backend menu) or RESET key is pressed, the keyboard controller transmits a key code greater than `0xf0`. This case is caught in the lines 11 to 13. If such an key code is transmitted, the emulation function is terminated and the key code is placed into the “`emulation_state`” variable. The main loop then checks if a further action needs to be done. This approach is more responsive than placing the key code from the main loop during two emulation execution calls of `exec()`.

3.3.4 Bringing software into the emulator (module “DSK I/O”)

The module to read disk images is very important, since it allows to load programs and other content into the emulator and run them using the emulator. As described before it uses the SD card to load the program data.

The old 5¼-inch floppy disks can be read out by an old computer with such a floppy drive and written into a single file, called *disk image* (with the file extension .dsk)[19]. These disk images may then be placed on a microSD card using the computer. The microSD card is inserted into the microSD card slot of the emulator and simply accessed through the backend menu. The text based assistant will guide the user through the lists of disk image files and programs on the disk and load the data from the disk image into the emulator.

To accomplish this mission, one needs to know more about the format of Apple][disk images. Unfortunately, no literature about this topic was found. So the format was discovered using the learning-by-doing technique and the “DiskBrowser” program from

19 Downloading them from the internet without owning them or having a permission from the author might be illegal.

Denis Molony[20]. Using the program, some sample disk images and a basic knowledge of the →FAT file system it is possible to find a way to the structure of the disk images.

Because of the fact, there exist several different disk formats for the Apple][(DOS disk, ProDOS disk, … etc.) disk, only the simplest (DOS disk) is used further to keep things simple.

Also no source code is shown bellow, because it would confuse due to the fact that it is really long and mixed with the code for the user interface. Instead the general structure is explained.

Structure of the disk images

A disk image file has a static size of 143,360 bytes. The image is segmented at a block size of 256 bytes into 35 tracks with 16 sectors each track. So the entire image file is divided into 560 blocks with a block size of 256 bytes. Dealing with the blocks in the right order and interpreting the values correctly will allow to read "files" or program data from the disk image. The structure of the disk image resembles the FAT (File Allocation Table format).

Figure 7 shows a graphical overview over the structure. On a simple DOS disk, the first block to read is the VTOC block. It contains many information about the disk, its memory state, the last allocated block and probably some other data. Beside this information the locations `0x01` and `0x02` of this block contain the track and sector of the first CATALOG block on the disk.

Furthermore, the VTOC contains some unique data to identify a "valid" DOS disk, like locations: `0x34` and `0x35` – the maximum number of tracks (`0x23` = 35) and sectors on the disk (`0x10` = 16) – and location `0x37` – the number of bytes per block (`0x01` = 1 → 256 byte). Although the track of the first sector needs to have the value `0x11` (17) at location `0x01` on this block.

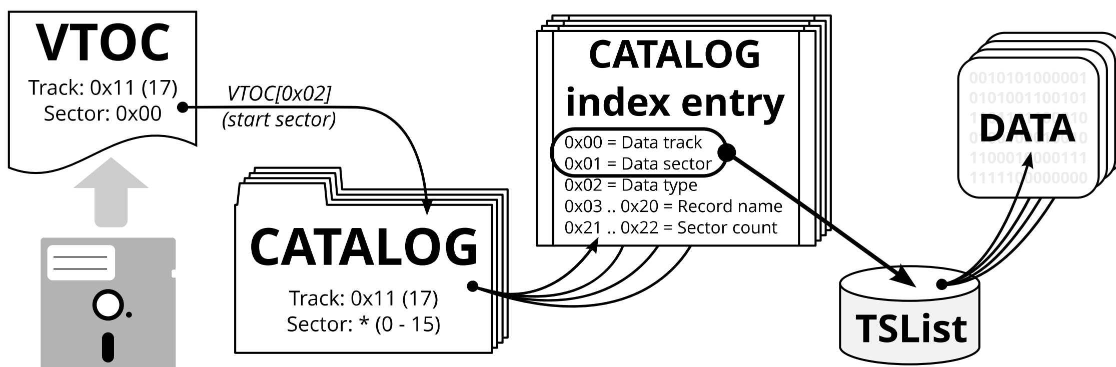


Figure 7: general structure of Apple][disk image files (Apple DOS disk format).

If this data is checked one can advance to the CATALOG entries. As the name intends, these blocks contain index entries with information about the files on the disk and where to find those files. The bytes `0x01` and `0x02` of a CATALOG entry point to the next catalog entry. If they are both zero, there is no further CATALOG entry. Every CATALOG entry contains seven index entries. An index entry has a length of `0x23` (35) bytes, starting at `0x0b` (11) in the CATALOG block.

20 Unfortunately, Denis Molony's website is not existing any more. Only the program can be found at Asminov's Apple][archive: ftp://ftp.apple.asimov.net/pub/apple_II/utility/DiskBrowser.jar.

An index entry represents the "header" of a file with control information. It contains the name, length and other file attributes. On index `0x02` of the entry, the file type is specified whereby the lower nibble identifies the file type and the 4th bit of the upper nibble is set if the file is locked. Interesting file types are: `0x01` for Integer BASIC programs, `0x02` for Applesoft BASIC programs and `0x03` / `0x04` for binary data. The first two bytes (`0x00` and `0x01`) of the index entry point to the track and sector of the TSList.

TSList stands for "Track-Sector-List". This is a simple list of all data blocks assigned to the file represented by this CATALOG index entry. Beginning at location `0x0c` in the TSList block a track and sector pair is listed, which points to the attendant data block. If both, track and sector, get zero, there is no more data to read. So obtaining the file data is a simple loop over the TSList, reading out the intended blocks.

Placing a program in the memory

Now that we know how to read out data from the disk image the next question is how to place this program data in the emulator memory in order to get the program running. The three file types described prior have a preamble in the beginning of the data containing the length of the data followed by the start memory location in the Apple][memory. So the disk loader knows at which point the data should be placed in the main memory.

But that alone is not enough to get it running in case of BASIC programs. They need further adjustment of other "configuration" memory locations:

- **Integer BASIC** – this program data does only have a preamble with the length of the data. The location to place it in the main memory can be calculated: the length of the data is subtracted from the highest (user-)memory location. This is the location for the first byte of the Integer BASIC program to load. Now only the "`HIMEM`" variable (at `0xca` and `0xcb` at the Apple][main memory zero page), pointing to the highest memory location must be set to this base address. After this highest memory location the tokenized Integer BASIC program lines from the disk are placed. If a new line is added, everything gets shifted back, the "`HIMEM`" variable is decremented and the new line is placed at the upper end of the memory [25].
- **Applesoft BASIC** – the preamble contains the length of the data and the location where to place it in the main memory. Several locations need to be changed in the zero page of the Apple][memory [38] (p. 140):
  - program start position at `0x67` and `0x68` (normally `0x0801`)
  - simple variable space start position at `0x69` and `0x6a` (one or two memory locations after the end of the program)
  - start position of the array space at `0x6b` and `0x6c`
  - pointer to the end of the numeric storage at `0x67` and `0x68`
  - start position of the string storage at `0x69` and `0x70`
- **binary data** – the preamble contains length and location in memory. This data must simply be placed starting at the location in the main memory.

Once the data is placed in the memory and all locations are adjusted properly, the emulation can run the program.

Usage in the emulator

If one has entered the backend menu of the emulator runtime environment, there is the option of loading data from a disk image from the SD card. Before selecting this option the user needs to make sure that the SD card is inserted into the SD card slot. Then the user can browse through the disk images and load a program from this image (see also “3.3.7 Emulator backend UI insights“, p. 62). Please note that the emulator must be set to the right BASIC required for the program on the disk. Due to the fact that the BASIC version can only be changed through an emulator reset, the data will be lost. Switching automatically to the correct BASIC isn't possible, because the emulator needs to run and stay inside the BASIC prompt in order to inject a BASIC program. On emulation resume the Apple][BASIC would “boot” and drop all data in the memory.

3.3.5 Hibernation feature (module “State I/O”)

The last feature to implement, in order to complete the feature list, is the hibernation feature. It allows the storage of the current emulator state to a non-volatile memory including the 12K emulated main memory of the Apple][system. Once the user wants to go back to this state he can simply load the data and continue emulation at this point.

As non-volatile memory an EEPROM is used. The software module provides only two public functions:

```
1    void save_state();
     unsigned char load_state();
```

Listing 16: public functions of the state I/O module.

The function “`save_state`” shows a text based save dialog, displaying the slots and their names. It lets the user select a slot to store the current emulation state to and give it a name with a string input prompt. The opposite functionality is done by the function “`load_state`”: the list of slots is shown to the user and he needs to select a stored state. The state is loaded and the emulation resumed.

The used EEPROM has a capacity of 128K. A single state has a size of 12K for the Apple][memory and some additional bytes for the global variables defining the state of the emulation. Furthermore the EEPROM is divided into pages of 128 bytes. Since the memory cells of the EEPROM have a definite life time and due to some hardware restrictions, writing one byte ages an entire page of 128 bytes. So in the following, entire EEPROM pages are written and read.

The 12K of the Apple][main memory are exactly 96 EEPROM pages. For storing a complete state with control information 97 EEPROM pages are used. This gives 10 places or “slots” where states can be saved. The left memory of approximately 8K remains free, because there is no further usage for it.

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
0x00	`0x2a`	state	cyc	a2p	gm	keyl	accu	regX	regY	regP	SP	PCL	PCH		Name	
0x10	Name															
...	Unused space of the first EEPROM page															
0x70																
0x80	12K of Apple][emulator main memory (EEPROM pages 1- 97)															
...																
0x3070																

Table 8: memory map of a single emulator state.

A single page entry has the structure, as shown in table 8: the first value is the magic number of this “file”. The next six values are the global emulator variables, storing the graphics mode (`gm`), key latch (`keyl`), emulator state (`state`), unused cycle count (`cyc`) and the currently used ROM (`a2p`). These are the global “management” variables of the emulator. The following six variables are the 6502 processor registers, defining the state of the 6502 microprocessor. The name (`0x0e` to `0x19`) is only used to let the user identify the state by a name he entered on save.

As seen the entire state of the emulator is stored in the EEPROM enabling the emulator to completely restore an older state from “hibernation”. The ten available slots are distributed evenly over the 128K EEPROM memory.

3.3.6 Sound output

The sound output was not mentioned until here, but this feature is quite simple because the Apple][did only use a standard speaker. When the speaker memory address was referenced the logical level of the speaker was changed performing a short click. Doing this process very often and at a specific frequency allows the output of (basic) tones. This design was inherited using a piezo buzzer on an output pin of the microcontroller. The code is implemented directly in the memory write macro, as seen in the code excerpt of memory read macro in section “Memory emulation“ (p. 44).

3.3.7 Emulator backend UI insights

Up to here all features of the feature list for the emulator runtime environment are implemented. All these features provide a sophisticated backend menu to manage the emulation. To enter the backend menu from the emulation, one needs to press “SHIFT + RESET” or “CTRL + RESET”. The following figure shows the backend user interface.

The main menu, as shown in figure 8, is pretty simple: the user only needs to press the number of the desired option.

Figure 9 shows the save dialog of the hibernation feature. On save or load the names of all ten slots from the EEPROM are listed to let the user decide which slot he wants to use. When the user selects a state (not shown) by typing the number and pressing “ENTER”, he is prompted to enter a name for this emulator “image” to identify it later on (only in the case of a save dialog). With the next “ENTER” the user confirms and the data is saved. Note that one can exit all actions by pressing the “ESC” key.

Figure 10 and 11 show the user interface to select a disk image from the SD card and load a program from this image. First a list of all image files on the root directory of the SD card are shown. The user can enter the number and press "ENTER" to select an image file or browse through the list by pressing "→", if there are more files than a screen can show. Any other input will quit this dialog. When the user selects an image file the contents of the disk image file are shown, if possible. This also is a long list, which can be passed through with the same advance. By entering the number of the program data to load and pressing "ENTER", it is loaded into the memory of the emulator. Please note that the emulator must be ready for this. That means that – in case of a BASIC program to load – the emulator must be set to the correct ROM and in the BASIC prompt.

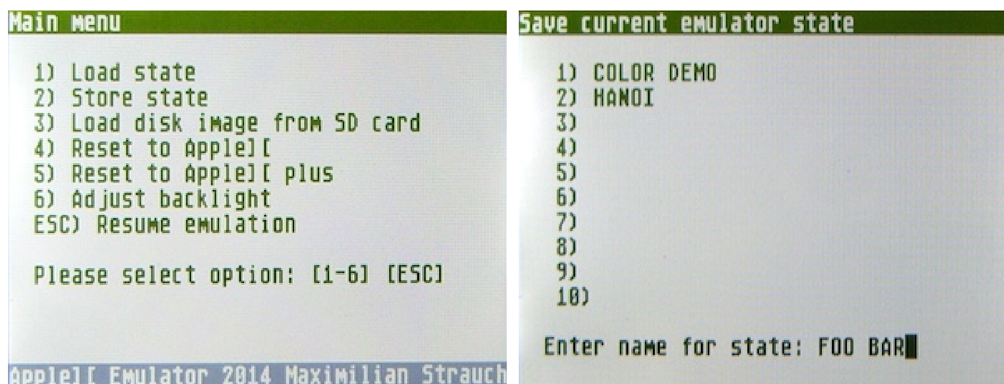


Figure 8: main menu of the emulator backend menu. *Figure 9: save state dialog, showing all slots and the input prompt.*

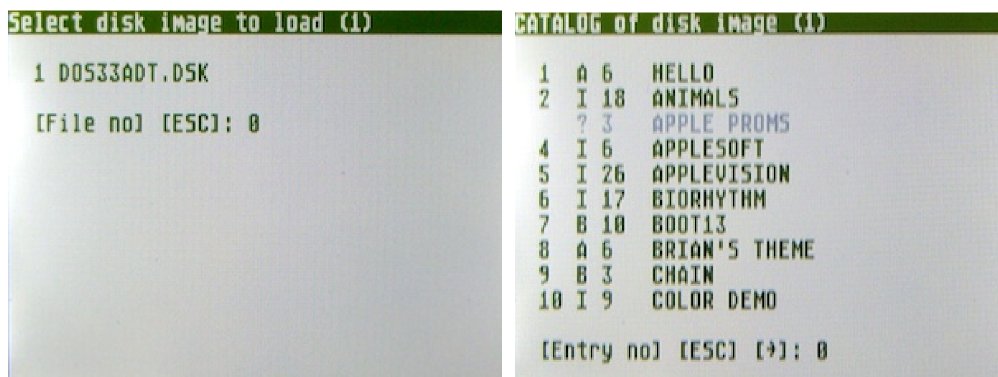


Figure 10: dialog to select a disk image from the SD card. Currently only one disk image is in the SD card's root directory. *Figure 11: first page of the CATALOG of the disk allowing to user to select a program to load.*

Beneath the program number in figure 11 for every line more information is shown: the type of the program (A = Applesoft BASIC, B = binary data and I = Integer BASIC), the length in sectors and the name. If an entry is colored gray, it cannot be loaded into the emulator memory due to the fact that it is to big for the emulator memory or has an unsupported type.

Chapter 4: Hardware implementation

After implementing the software for the handheld Apple][emulator device, this chapter covers the implementation of the hardware. It will also cover some software aspects which are related very tight to the hardware. The way towards this final structure is not shown, because that would go far beyond the scope of this document.

The concept of the hardware realization is very simple: the final result should be a battery powered, portable Apple][emulator device which runs the emulation on a single AVR microcontroller using as less peripheral parts as possible.

As carried out in the chapters before the complete system is split into two parts: the emulation microcontroller ,which runs the emulation, performs display output and contains the backend menu to manage the emulation. The other part is the keyboard controller, managing the custom keyboard built from scratch.

The following sections will first explain the emulation microcontroller and its hardware design and then the keyboard microcontroller. Finally a bill of materials will list all the components which are required and the schematic shows the “circuit”.

4.1 The “emulation” microcontroller

The microcontroller dedicated to emulate the Apple][system, including input and output, needs to have a very high performance to match the speed constraint of the emulator. Browsing the different types of Atmel AVR microcontrollers, the ATMega 1284 was chosen, because it matches all requirements as best as possible. It provides the following features [39]:

- (up to) 20 MHz clock speed (maximum for AVR 8 bit microcontrollers of the ATMega series).
- comes in a DIP package and is easy to mount for the purpose of a prototype.
- provides 128 KB program memory.
- has a SRAM of 16K (e.g. in contrast to the ATMega 2560 which has 256K pro-

gram memory, but only 8K static RAM and is a surface mount component).

- comes in a DIP-40 package with with 32 usable pins as in- or output pins to interface the peripheral components (display, keyboard, etc.).
- provides other hardware features, which will come in handy later like SPI, TWI (IC^2), UART / USART for serial communication and hardware timers.
- runs with standard +5V logic level.

4.1.1 Pinout and pin mapping

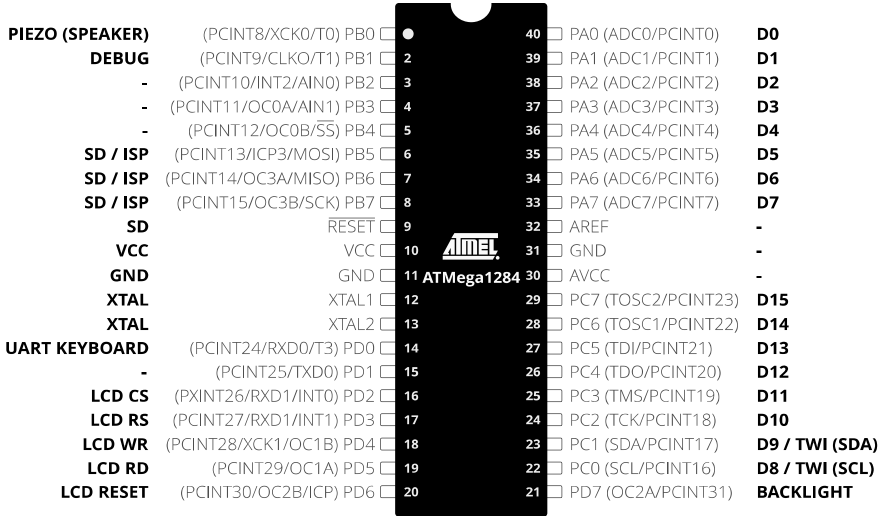


Figure 12: pinout of the ATMega1284P-PU used for the emulator controller along with the used mapping (bold).

Figure 12 shows the pinout of the ATMega 1284 [39] with the pin mapping for this project. The pins are assigned as follows:

- port A and port C form the 16 bit data bus to the display. Using two entire ports for the display makes it easy to write the data to it, because no data needs to be shifted out or transformed in any other way. Despite the communication between the display and the ➝MCU being bidirectional by design, in this case only an unidirectional model is used.
- port C0 and C1 are also related to the TWI hardware providing a two wire communication interface. Since the EEPROM uses this TWI interface to communicate, these pins are used to perform this task. Because of the fact that these two pins are assigned with two tasks only one can be performed at a time (display output or TWI communication).
- port D7 drives the backlight brightness of the display by using ➝PWM. By adjusting the generated square wave the brightness of the backlight can be increased or decreased. This has not only aesthetic reasons but is also used to extend the durability of the battery. The background light consumes the main part of current (around 165 mA on maximum brightness). By driving it with pulse width modulation, the power consumption can be reduced to around 70 mA at the darkest setting and thus extending battery life. It also shows a nice usage of the hardware timer.

- port D2 to D6 drive the display controller. These are the control lines, controlling the display driver IC.
- port D0 (RXD) and port D1 (TXD) are used for serial communication with the keyboard controller using the hardware UART interface. Currently the TXD (transmit data) line is not used but it is reserved for later use and could serve as a debug output during further development. The keyboard controller only sends data, so only the RXD pin is needed.
- XTAL1 and XTAL2 connect to the 20 MHz quartz generating the system clock signal.
- VCC and GND are connected to the power supply.
- port B5 to B7 and $\overline{\text{RESET}}$ are used for the AVR →ISP to program the microcontroller. The pins are connected to a standard 10 pin box header to simply connect a programming device [40] (p. 7, figure 3-1).
- port B4 to B7 are also used to interface the SD card through the SD card library, because these pins belong to the hardware SPI interface.
- port B1 is a simple debug output pin. Every time the emulation performs 1.023 MHz of the emulated MOS 6502 microprocessor the logical state of this pin is toggled and so a speed measurement can be done very easily (see p. 91).
- port B0 serves as the speaker output. This digital pin is directly controlled by the MOS 6502 emulator and its output state is toggled every time the speaker memory location of the Apple][is referenced.

Using this pin mapping only two free pins remain (port B2 and B3), which do not allow the implementation of any further features like annunciator ports, utility strobes, cassette input or outputs or game strobe inputs as shown in table 5 (p. 20).

From the side of the emulator microcontroller the serial connection to the keyboard needs no further adaption. It was sufficiently explained in the software section (see section “Keyboard input (module “Keyboard”)“, p. 56) and there is no need for more hardware. Only a wire connecting these two pins of the emulator microcontroller and the keyboard microcontroller is needed. Also the speaker output (port B0) and simple debug output (port B1) are single port outputs and set by previously discussed code segments.

4.1.2 Interfacing the display

The display used for the prototype is a standard 3.2 inch TFT LCD display with a resolution of 320 pixels by 240 pixels. It uses the Solomon Systech SSD 1289 display controller. Because of the fact that the display data bus is 16 bit wide, the display works with 16 bit RGB colors. Thereby the RGB is split into five bits of blue, six bits of green and five bits of red. Beneath the data bus, there are the five display control lines, which control the behaviour of the display [41]:

- $\overline{\text{CS}}$ (PD2) stands for “chip select” and is active low: if this line is tied to ground, the display accepts commands. If this line is hold high, no changes can be per-

formed. This is useful because the TWI uses two pins of the display data bus and using this control line one can prevent changing some data on the display while using the TWI lines.

- RS (PD3) or “register select” sets the selected target to read or to write from. The display not only has the graphics data RAM for the pixels and their color, it also has some registers controlling many other features of the display. Setting this line to low indicates that the target of the next operation or command is the RAM; setting this line to high indicates that a register is the target.
- WR (PD4) and RD (PD5) are determining if one wants to read data from or write data onto the display. The states are [41] (p. 21, a)):
  - WR := 0 and RD := 1 in order to read from the display.
  - WR := 1 and RD := 0 in order to write to the display.
  - the other two configurations are forbidden.
- $\overline{\text{RESET}}$ (PD6) is low active. If this line is pulled to ground, the controller resets until this line is set to logical level high.

The process of setting up the display consists of toggling the $\overline{\text{RESET}}$ line to reset the display controller and its data registers and then writing various data to the (control) registers of the display to setup the display for usage. Those data which are written to the register are responsible for setting up properties like color model, inactivation of sleep mode, screen resolution, gamma correction and a lot more properties to (de-) activate the different features of the display. This initialization data is quite standard and was adapted from the example programs provided by the distributor of the display.

Once the display is ready to run the next challenge is to put data on the screen. There are two possible ways to do that: setting a single pixel or using the “burst” mode. Setting a single pixel might sound like the right way, but it is not. If the entire screen should be filled with a color (clearing the display), 76,800 pixels must be send to the controller with their color. By setting every pixel, there is the need to send the two color bytes, the location of the pixel on the screen with two bytes and toggle the WR line. This are six write operations to the I/O pins of the microcontroller, which must be performed 76,800 times. This process takes a long time due to many redundant data transmissions (color, pixel positions and control lines).

Fortunately the “burst” mode works in another way: once a region of the display needs to be redrawn (to clear it or show a character symbol), multiple pixels must be set forming a rectangle or window to redraw. In this mode, the driver sends at first the start location (P_S) on the screen and then the end location (P_E) on the screen.

This spans a window region on the display to update as seen on figure 13. This window is going to be filled. Next the display writes the 16 bit color data on the bus for the first pixel. Every time the WR line is toggled (to low and then to high) a pixel of this color is added and drawn to the window. The arrows in figure 13 indicate the direction in which the “virtual” pixel-pen is moved. Internally, the display controller has a simple pointer to the graphics data RAM which is incremented in a certain way after every WR

toggle. This direction can be adjusted in the control registers. To display an image with multiple colors the driver needs to change the color data on the bus before toggling the WR line. This allows the controller to draw complex images in one “piece”, omitting the calculation and transmission of the pixels.

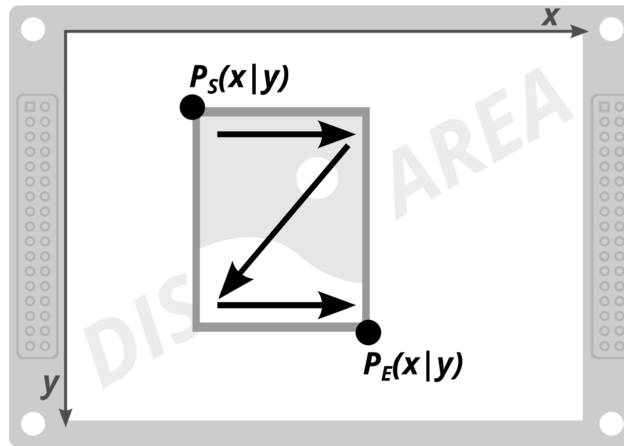


Figure 13: illustration of the "burst mode" defining a "window" inside the display area, which is then filled with pixels line by line in the direction indicated by the arrows.

Using this mode, the function to clear the display with a specific color can be written in the following way:

```
1    void lcd_clear(unsigned short color) {
       DISPLAY_WAKE;

       unsigned short i;
5
       _lcd_write_address(0, 0, 319, 239);
       _LCD_DATA(color);

       for(i = 0; i <= 0x9600; i++) {
10       _PULSE_WR();
         _PULSE_WR();
       }

       DISPLAY_SLEEP;
15   }
```

Listing 17: display function to fill the display with a solid color.

First, the display is selected through $\overline{\text{CS}}$. It is now ready to take and execute commands. Then the size of the window or start and end points are transmitted (line 6). After this, the desired color to fill the display with is directly written to the bus (line 7) and the loop from line 9 to 12 “adds” the pixels to the window. Since `i` is an unsigned short, it ranges from 0 to 65,535. But there is the need to write 76,800 pixels, which would go beyond the range of the counter. By pulsing the WR line twice in the body of the loop, two pixels are set and the loop only needs to iterate over half the number of pixels ($38{,}400_{10}$ = 0x9600). Finally, in line 14, the display is “released” to ignore further data on the control and data lines. Using this technique, it takes a small amount of time to fill the screen.

Using the exact same scheme the functions “`lcd_apple2_text`” (Apple][character and low resolution block draw function) and “`_lcd_put_ascii`” (draws an ASCII text

character in a 16x8 pixel font) from the display driver (`display.c`) can be created. Simple time measurements give the total amount of around 36 ms for drawing an entire Apple][screen with 960 characters. Be aware of the fact that this time is slowed down by the function to draw the display (“`draw_apple2_display`” in `main.c`), because of the fact that the global memory array must be accessed 960 times and the global variable for the graphics state.

The “fonts” are stored as black-and-white images in an array. Every byte stores 8 pixels. For the 5x7 pixel Apple][font, 7 bytes form an image of a character. These font arrays are stored in the program memory to save storage in the SRAM. The same applies for the enhanced 8x16 pixel font of the backend menu.

4.1.3 Talking to the EEPROM

An EEPROM (non-volatile) is used to store the states of the Apple][emulator and provide the hibernation feature. EEPROM memory cells have a limited lifetime. The used 24LC1025 EEPROM with 128K memory from Microchip has a life time of 1.000.000 write cycles per byte. This is only the minimal number of write cycles it will sustain. If the bytes lifetime is passed the data written, might not be stored correctly anymore [42].

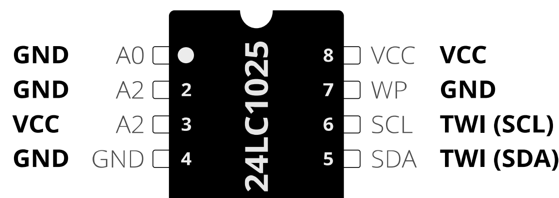


Figure 14: pinout of the 24LC1025 EEPROM from Microchip with pin mapping (bold).

The 24LC1025 comes in an DIP 8 package and uses the TWI communication interface to talk to other devices. Figure 14 shows the pinout of the IC [42]:

- A0, A1 are used to chain up to four EEPROMs together. This allows a memory extension from 128K to a maximum of 512K. Because of the fact that for this project 128K are sufficient, these pins are tied to GND.
- A2 is an non configurable chip select and must be tied to logical high [42].
- VCC and GND are connected to the power supply.
- SDA and SCL are the two wires of the TWI interface. SCL serves as the clock line and SDA is then used to transmit the data bitwise.
- WP prevents writes to the EEPROM if this pin is logical high. Since we do not need this feature, this pin is also tied to GND.

So reading and writing to the device is done by the TWI interface. The TWI is a protocol that was developed by Phillips with the name IC^2 at the beginning of the 1980s. It uses (only) two wires, the serial data line (SDL) and serial clock line (SCL) to transmit data between devices. This protocol allows up to 127 devices on the same two wire bus. In most cases there is one master and multiple slaves, which communicate to the master.

Although a multi master TWI is also possible [43]. To communicate, the master occupies the bus and sends the address of the device he wants to talk to. Then he sends the information whether he wants to write or read the device[21]. The data will be transmitted by the master (if he wants to write) or the client (if the master wants to read). Finally the master releases the bus. The protocol has more mechanisms like START and STOP conditions to ensure a stable communication. Explaining the entire TWI protocol would go beyond the scope of this document and is omitted here [44].

In order to operate proper the two TWI lines need to be pulled up through a pull-up resistor [44]. In this case 10kΩ are used (see also recommendation in [42]). The TWI bus can operate at different bus speeds. In this project the hardware TWI module of the AVR is used and set to a maximum rate for the EEPROM: 400 kHz [42].

The “State I/O” module relies on the “EEPROM” module (files `eeprom.c` and `eeprom.h`). These source files manage the hardware TWI module of the ATMega 1284p and provide two handy functions to read from or write to the EEPROM. Because of the fact that the TWI lines are used by the display data bus, only one of these two components can be active at a time. If the TWI hardware of the AVR is turned on the port settings of the pins are overwritten and adjusted for the TWI. After turning off the TWI module, the overwritten pin settings are set back to the initial ones. So if the program starts the execution of code in one of the EEPROM functions the TWI is turned on and at the end of the function turned off. This ensures that the display can then continue work normally. Since the microcontroller is not able of multitasking and no interrupts are used, there is no possibility that display methods are called during an active TWI.

The actual implementation of these functions is of no interest, because it consists only of sending data and receiving the result. So this will not be discussed any further.

4.1.4 The SD card and ISP connectors

As described the pins $\overline{\text{RESET}}$ and port B5 through B7 are used for the ISP interface. The ISP or “in-system-programming” is an interface used by programmers to bring the compiled program from the computer onto the AVR microcontroller. Therefore the hardware SPI module of the microcontroller is used (lines port B5 to B7) along with the $\overline{\text{RESET}}$ pin. The idea behind this interface is to program the microcontroller in its target circuit without removing it, as described in AVR application note 910 [45]. Because of the fact that this interface relies on the hardware SPI interface of the microcontroller, other SPI peripherals could disturb the ISP. Once the programer wants to program the microcontroller, it pulls down the $\overline{\text{RESET}}$ line. Then the AVR microcontroller automatically sets all lines to input with pull-ups disabled to receive the program data [45] (p. 2). Once the programming is finished, the programer releases the $\overline{\text{RESET}}$ line and the microcontroller runs the flashed program. The data put on the SPI lines, which may be connected to another peripheral device, does not affect the other device, because the chip select line is inactive and so the slave ignores any data.

Like the TWI, the SPI protocol is used to transmit data in a serial way between two devices. It consists of three lines: MOSI as “master out – slave in”, MISO “master in –

21 Apparently the device address consists of 8 bit. If there can up to 127 devices, one bit in the address is left. So the LSB is used to determine, if the master wants to read or write [43] [44].

slave out" and SCK as the serial clock. A forth line, called $\overline{CS}$, is then used to activate a slave [46] (see figure 1 on page 1). This allows to connect any number of slaves, only limited by the pin number of the master for the chip select pins.

In contrast to the TWI protocol the SPI has no further conditional statement (START or STOP condition) and only shifts the data out. Furthermore the TWI can operate at a maximum of 400 kHz (fast mode) and 3,4 MHz (high speed mode) [43] [44], whereas the SPI module can operate with more than 10 MHz [39]. Another difference is, that the SPI receives a byte when it sends and vice-versa: when data are shifted out by the master through the MOSI line, data from the MISO line get shifted in from the client (see figure 1 on page 1 of [46]). Like the TWI protocol a deeper explanation could fill an entire document. So other details are omitted here.

As put out before a SD card is connected to the microcontroller to provide access to disk images. This SD card can be interfaced through an SPI interface. As shown in the pinout (p. 65), the SD card is connected through the hardware SPI port to the ATMega 1284p on port B4 to B7. Dealing with the SD card and parsing the content on the card in order to access files on the FAT16 file system of the SD card is performed by the "Petit FAT File System Module" library [37].

On the hardware side, the SD card operates on a voltage level of 3.3 V. As pointed out before the emulation microcontroller works with 5V, since this voltage is required by using the 20 MHz crystal. A linear voltage regulator is used. It regulates the voltage from the 5V power source down to the required 3.3V and the ground line is common (another regulator is also used to regulate the raw battery power down to 5V). There are four other lines to care about:

- **SS** (socket select), **MOSI** and **SCK** have one property in common: they are output lines of the master and input lines of the SD card (slave). They are more complicated because the outputs of the master and can rise up to 5V which will destroy the SD card. So in this case a simple voltage divider with values of 1.8 kΩ and 3.3 kΩ is used to reduce the voltage level [47] (p. 122).
- **MISO** as input of the master and output of the SD card (slave). This line can be connected directly, because it is an input line of the emulation microcontroller and does not get up to 5V, since the SD card operates on 3.3V only and can output this voltage at its maximum. Due to the fact that the ATMega 1284p recognizes voltages as high if they are above 0.6 × VCC (equals 3V or more for VCC = 5) this line needs no further treatment [39] (p. 323).

By this circuit and all the thoughts before, the SD card can now be accessed through the simple API, provided by the PetitFS library. And with this last component the hardware interface setup of the emulator microcontroller is complete. So the next aspect of the hardware implementation is the keyboard controller.

4.2 The "keyboard" microcontroller

As shown before, the keyboard microcontroller is responsible to listen for keyboard input and send it to the emulator microcontroller via UART.

By design this structure allows an easy change of the input keyboard method, because of the abstraction between the keyboard controller and the emulator microcontroller. In the following sections the custom-made keyboard will be explained. It is fairly easy to change the program of the keyboard microcontroller to interface another keyboard, like a PS2 or USB keyboard or even a touch screen display with an on-screen keyboard, without the need of updating the emulator microcontroller or being restricted through limitations of pins or CPU time on the emulator microcontroller.

The requirements for the custom-keyboard for this project are: it should be (1) small enough to fit in a portable device case, it should be (2) portable by itself (unlike a standard keyboard) and finally it should contain (3) the exact keys and keyboard layout of the Apple][.

For the keyboard controller an Atmel AVR ATMega 8 microcontroller was used [48]. In contrast to the ATMega 1284p used for the emulator this microcontroller has less features. The main features used are:

- comes in DIP 28 package, providing 23 programmable I/O pins.
- with 1K of static RAM.
- with 8K of program memory.
- with external quartz up to 16 MHz and with internal quartz up to 10 MHz.

The little amount of static RAM and program memory are not a big concern since the task of this component is very clear and simple. The external quartz is not used in order to save I/O pins for the keyboard matrix. The internal quartz is used at a speed of 10 MHz.

4.2.1 Pinout

Figure 15: pinout of the Atmel AVR ATMega 8 microcontroller with pin mapping for the keyboard controller (bold).

Figure 15 shows the pinout along with the port mapping for the keyboard controller [48] communicating with the emulation microcontroller and scanning through the key matrix. The port assignments are:

- port B, port D5 to D7 and port C4 and C5 are used for the 13 keyboard matrix columns output (order pointed out in the pinout).
- port C0 to C3 for keyboard row inputs.
- VCC and GND for power supply.
- $\overline{\text{RESET}}$ connected to the reset pin of the ISP. Despite this microcontroller has no dedicated ISP pin it is useful to reset both microcontrollers together to put them into “sync” although this is not necessarily needed.
- port D2 for software driven UART transmit as uplink for the scanned key codes to the emulator microcontroller.
- ports D0, D1 and D4 are reserved for later enhancements to connect a PS2 keyboard. These ports are related to the hardware USART module of the micro-controller, but this feature is not implemented yet.

So the port D3 only remains free without usage yet.

4.2.2 Keyboard switch matrix design

As known from the beginning of this document, section “The keyboard“ (p. 24), the Apple][had 52 keys. To keep hardware design easy the keyboard was interpreted as a matrix consisting of 13 columns with four rows. Whereby the last row, containing only the space key, is moved to the fourth row and fills it up to 13 columns. So there are four rows with 13 columns each.

Scanning the keyboard is now very simple: all columns are passed through and pulled to logical high level and the other twelve rows are pulled down to ground, so that only the current column has the state logical high. By iterating through the row inputs and watching out for rows which are at logical high, the microcontroller can exactly identify the key which was pressed because he knows the column put high and the row which is high. These two information identify a single key in the matrix like a point in a coordinate system is identified by x and y. In order to get this working stable the rows (which are inputs) must be tied to a defined state. So they are tied to ground using a pull-down resistor of 10 kΩ. By performing this action very often, the microcontroller can watch the state of the keys faster than any human can press.

So if a key is detected as pressed, the row and column position is translated by the simple formula “index := row × 13 + column” to an index of a linear array and the Apple][key code is simply read from this array. If a modifier key like CTRL or SHIFT is pressed, the index is looked up in another array containing the right key codes for this modifier key[22].

But there is another problem called “debouncing”. The keys used for the custom-keyboard do toggle their state when they are pressed for a very short amount of time until they make a steady connection. If one considers a simple circuit containing a toggle switch and an LED this effect is present, but it is so short (normally less than a fiftieth part of a second) that the human eye cannot recognize it. Using an oscilloscope one can see this effect. This effect will distort the pressed key: the key controller will

22 The key codes can be found in [21] on page 7 in table 2. Please note that all key codes for ← and → are interchanged.

recognize multiple key presses and send them to the emulator instead of recognizing a steady key press. The solution is very simple: after the first time of detecting a key press the microcontroller waits for a debounce time of some milliseconds before he rechecks the state of the switch. If the switch is already pressed he can recognize it as a key press. The delay time for the software debouncing is not fixed and varies from switch to switch.

This simple construct serves as a keyboard with controller. Note that the number of used I/O lines for the keyboard (currently there are 17 I/O lines dedicated to row input and column output) can be reduced with other means, for example to six by using an external IC: a "Johnson counter". It will pulse the column lines one at a time, but with only a reset and clock line. Then only the four row input lines are needed. To stay as simple as possible this design was not chosen. In fact the keyboard controller has enough I/O lines to perform this action in software and there is no need for an additional IC.

Please note further that key presses on the Apple][keyboard produce only one character independent of the time the key is hold.

4.2.3 Software UART transmit

If a pressed key was recognized and a key code was calculated the keyboard microcontroller transmits it via UART to the emulator microcontroller. The next time the emulator microcontroller checks the reception of a key code it can read the pressed key code. Because of the fact that the reception buffer of the emulator microcontroller is only one byte, key presses get missed if they are not read often. The reading of these key codes is done by the CPU emulation memory read macro (see "Memory emulation", p. 44) and so there is no need for further attention, because the emulator software takes care about this and checks it as needed.

Because of the fact that the hardware UART interface of the keyboard controller is reserved for the future connection of a PS2 keyboard and the ATMega 8 only has one hardware UART [48] the transmit function must be realized in software. This can be done using any digital I/O pin of the microcontroller and the following function:

```
1    void suart_put(unsigned char d) {
       unsigned char _tmp_sreg = SREG; // Get status register
       __asm__ ("cli" ::); // Interrupt disable
       SUART_PORT |= 1 << SUART_BIT; // Go high
5      unsigned short frame = (3 << 9) + (d << 1);
       while (frame) { // Send frame
           if (frame & 1)
               SUART_PORT |= 1 << SUART_BIT;
           else
10             SUART_PORT &= ~(1 << SUART_BIT);
           frame >>= 1; // Next bit
           _delay_us(SUART_BITL); // Wait bit time
       }
       SREG = _tmp_sreg; // Restore status register
15   }
```

Listing 18: software UART transmit function.

Sending a byte through UART is fairly simple: first, interrupts need to be disabled during the body of the function, forming an "atomic" section which is not interrupted. This

is done by saving the status register of the AVR and turning off the interrupts (line 2 and 3) and restoring the status register before return to reset the state of the interrupt flag (line 14).

Once this is done a UART "frame" can be assembled. Since this function transmits eight data bits, one stop bit and no parity bit a frame can be assembled in a very simple way: bit zero is always low and the start bit. Then the eight data bits follow and then the stop bit (always high) follows. After the stop bit the line remains high to indicate idle mode.

When the frame is assembled, (line 5) it simply needs to be shifted out through the pin (lines 6 to 13) and after shifting a bit out the microcontroller must wait the bit time (line 12). By knowing the processor clock speed and the desired baud rate, the waiting time can be calculated by the following formula:

$$\text{SUART}_{\text{bit length}} = \frac{\text{MCU cycle length } [\mu s]}{\text{(desired) BAUD}} \quad \text{(= macro "SUART_BITL").}$$

In listing 18 C macros are used to encapsulate the port and speed settings to be changed easily later on. They are not shown in the excerpt.

4.2.4 Possible disadvantages

Implementing a custom keyboard with tactile switch buttons brings some possible disadvantages with it, which are:

- the keyboard provides an unusual feeling of key press since it is small and the switches need a high pressure. To ensure an even better usage the USART was reserved to enable the connection of a PS2 in the future.
- every key has another debouncing behaviour. In the presented software all keys are detected using the same software debouncing time. This might lead to key recognition problems but the usage shows that it works fairly good.
- there are no security mechanisms integrated to handle the event of two input keys (not the special keys) being pressed at the same time. Simply the key with the highest row and highest column count gets selected because it is scanned last. For example pressing the one (1) and the space key at the exact same time the space key gets recognized. Since this is not a big problem and relies on the user there is no special handling need.

4.3 BOM

Since the several software and hardware aspects of this project are discussed up to here, the construction the handheld Apple][emulator can be started. The "construction", which consists of many steps from creating the schematic over purchasing the parts, soldering them together and building the final prototype takes a long time. All these steps are not described here, because this document is not a guide to build electronic projects. Instead only the result: the BOM, photos of the final result and the schematic are presented.

The bill of material (BOM) from table 9 lists all components used to build the prototype. The prototype cost about 35 € to build.

No	Part	Quantity	Total cost
For the emulation microcontroller			
1	3.2'' LCD touch sensitive display, 320x240 pixel resolution	1	13,18 €
2	Atmel AVR ATMega 1284p PU (DIP 40)	1	6,35 €
3	IC socket for DIP 40 (GS 40)	1	0,13 €
4	Standard quartz, 20 MHz	1	0,17 €
5	Resistor, 10 kΩ *(for TWI and controller reset)*	4	0,07 €
6	Ceramic capacitor, 100 nF *(for voltage regulators and ICs)*	6	0,38 €
7	Ceramic capacitor, 22 pF	2	0,23 €
8	Electrolytic capacitor, 10 nF	2	0,40 €
9	2x17-pin female header connector (BL 2X17G8 2,54) *(for the display)*	1	0,49 €
10	5V voltage regulator (MCP 1702-5002)	1	0,45 €
11	3,3V voltage regulator (MCP 1702-3302)	1	0,45 €
12	2 GB MicroSD card (with adapter)	1	3,95 €
13	Pin header 1x07 (right angled)	1	0,06 €
14	1,8 kΩ *(for SD card voltage divider)*	3	0,05 €
15	3,3 kΩ *(for SD card voltage divider)*	3	0,05 €
16	Boxed header, 10 pin angled (WSL 10W)	1	0,13 €
17	Piezo sound transducer (SUMMER BM 15B)	1	0,75 €
18	Serial EEPROM, 128K, DIP 8 (24LC1025)	1	2,55 €
19	IC socket for DIP 8 (GS 8)	1	0,04 €
20	NPN transistor (BC 548A)	1	0,03 €
For the keyboard microcontroller			
21	Tactile switch, 6x6x6 mm	52	0,82 €
22	Atmel AVR ATMega 8, P-DIP 28	1	1,85 €
23	IC socket for DIP 28 (GS 28-S)	1	0,09 €
24	Ceramic capacitor, 100 nF	1	0,07 €
25	Resistor, 10 kΩ *(for row pull-down)*	4	0,07 €
26	2x2-pin female header, angled (MPE 095-2-004)	1	0,24 €
General			
27	Stripboard (hard paper), 150x100 mm (H25SR150)	1	1,45 €
28	Battery holder, 4xAAA (HALTER 4XUM4-NLF)	1	0,45 €
29	Switch	1	0,14 €
30	Acrylic, around 150x100x2 mm	-	-
31	Stranded wires, different colors (around 7 m)	-	-

No	Part	Quantity	Total cost
32	Various screws and nuts	-	-
33	Solder	-	-
	Total cost:		**35,09 €**

Table 9: bill of materials for the prototype. The prototype cost about 35€.

4.4 The prototype

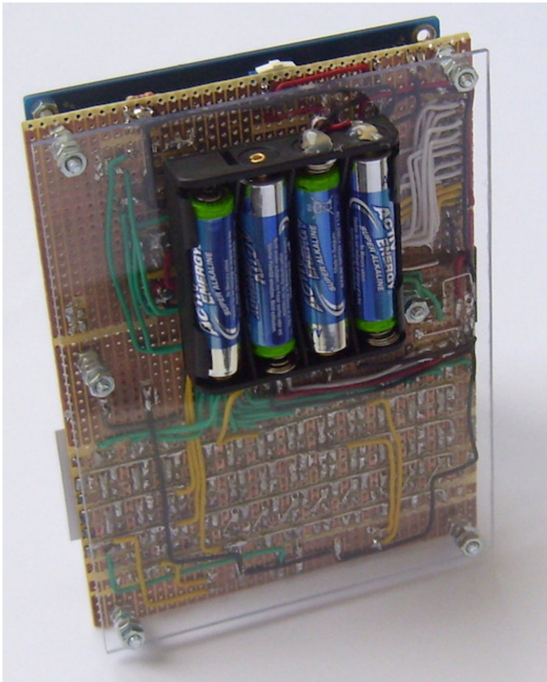


Figure 16: rear of the prototype with the battery pack and a view of the wiring.

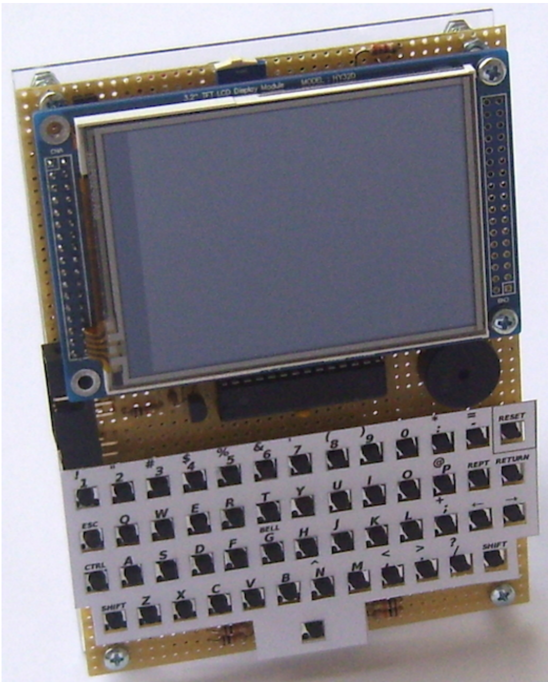


Figure 17: front image of the final (revision 3) prototype with keyboard overlay and all components.

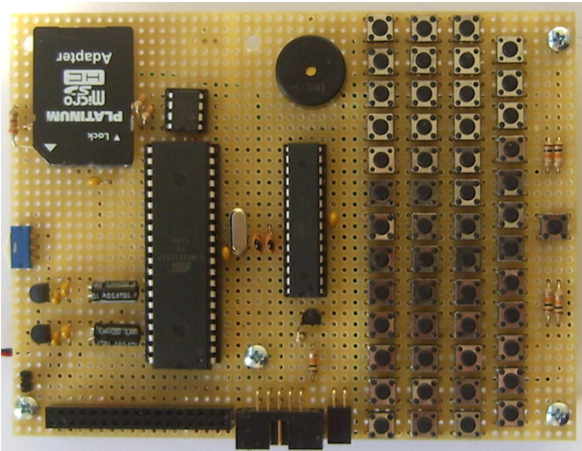


Figure 18: top view of the prototype without display and all other components attached.

Using all the components from the bill of materials and some skills in building and soldering the aimed battery powered, portable handheld Apple][emulator is created. As seen in the BOM the device is powered by four standard AAA batteries giving between six and seven volts at around 1200mA. Running the emulator with a mean display brightness (at a power consumption of around 105mA) the entire emulator takes 130mA (140mA with SD card in action). This should give a lifetime of around seven hours with a security deduction of 25% because of the dropping voltage. The voltage regulators will fail if the →dropout voltage is passed.

The previous images show the finally created prototype (revision 3) from different viewing angles with all its components.

4.5 Schematic of the prototype

The schematic of the prototype was made with the EAGLE Layout Editor 6.5.0 and is shown on the next page in figure 19. It is self-explaining with the information from the previous chapters.

There are still some parts of the electronic side of the project which are not covered in this documentation, because it would go beyond the scope of this thesis. All these not covered parts like the oscillating circuit or the voltage regulator setup have common “standard” solutions so that there is no need for explanation.

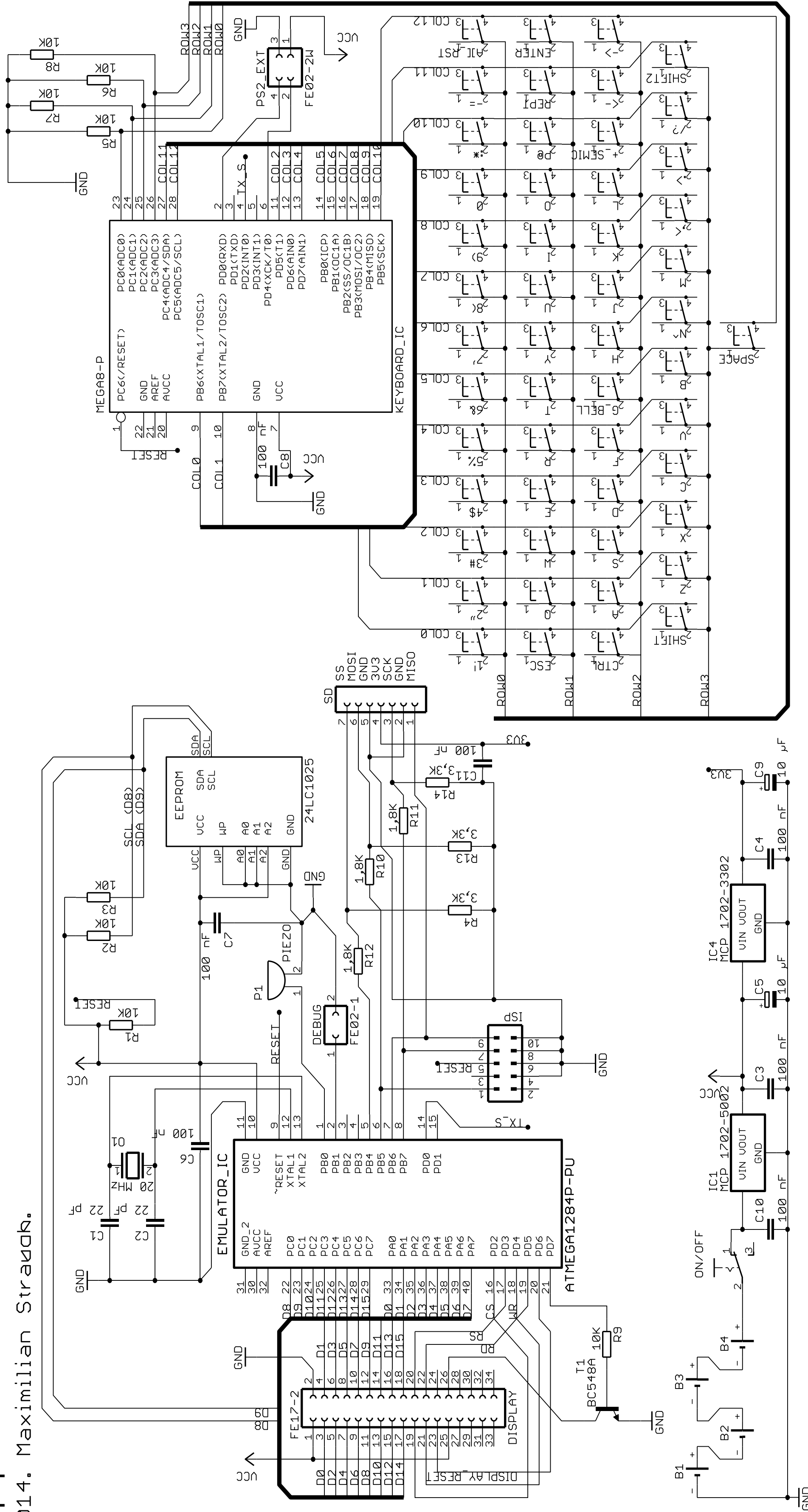


Figure 19: schematic of the entire Apple][emulator circuit with keyboard controller.

Chapter 5: Conclusion & Outlook

5.1 Conclusion

The target of this work is the development and implementation of a portable, battery driven handheld device which emulates the entire Apple][computer system. And to take it short this target has been reached successfully as the prior chapter(s) describe. So one can take a recap of the demand profile for this thesis from the beginning (p. 7) and spot the achieved requirements out of this profile:

- ✔ **implementation of a 6502 microprocessor without the decimal mode in C or assembly language** – as described in chapter 3 “Software implementation“ (p. 29) the 6502 microprocessor was successfully implemented in C and in AVR RISC assembly language. The assembly language was chosen for the final solution, because of its high optimization level and major speed improvement.
- ✔ **interfacing a TFT display with video RAM** – as described in section 4.1.2 “Interfacing the display“ (p. 66) a custom driver was written for the used display controller, utilizing some hardware features of the display to speed up the rendering of text to it by using the “burst” mode.
- ✔ **sketching a custom keyboard with controller** – to provide a “historical” accurate device a custom keyboard was developed with the original keyboard layout. See section 4.2 “The “keyboard” microcontroller“ (p. 72) for details.
- ✔ **realization of the Von Neumann architecture on the Harvard architecture of the microcontroller** – by implementing the assembler version of the 6502 microprocessor emulator, this requirement is satisfied: the structure of the Apple][system (Von Neumann architecture) was mapped to the microcontroller (Harvard architecture), for example the proceed of placing all ROMs in the program memory.
- ✔ **implementation of different memory accesses (RAM, ROM, mapped I/O)** – as described in section “Memory emulation“ (p. 44), this requirement is also implemented. There is no way around targeting a working Apple][emulator without distinguishing between different types of memory accesses. Besides the distinction between RAM and ROM is not necessarily needed, the Harvard architecture of the microcontroller forces a distinction because of missing static

RAM to hold the ROM data.

- ✔ **software loading possibility –** this is a very important feature in order to really use the created handheld device by running old programs which were written for the Apple][. In this thesis a microSD card is used (see section 3.3.4, p. 58 and section 4.1.4, p. 70) to bring software on the device. The backend menu lets the user browse through the disk images inside the root directory of the card and select a program to load.
- ✔ **buildup as a mobile handheld system –** as demonstrated on the photos in section 4.4 (p. 77) the result is a real working, battery powered, mobile handheld device based on the thoughts of all previous chapters.
- ✔ **documentation of the result –** for this requirement this entire document will serve as an evidence.

By satisfying all this requirements for the thesis there are even more features which were implemented, going beyond the scope of the demand profile. Those further features are:

- ★ **providing a hibernation feature –** in section 3.3.5 (p. 61) the feature of saving the entire emulation state to an EEPROM (non-volatile memory) was described. It enables the user to name and save the current state to one of ten slots and recover it later continuing emulation from this point.
- ★ **implementation of a text user interface –** a basic text user interface was designed to allow the structured input of numbers and strings along with menus and displaying everything through the display using a special, nice readable, 8x16 pixel font. This enables the device to be even more usable.
- ★ **the backend menu –** in order to load or store hibernation states or perform other configuration actions an emulator backend was added. By using the key stroke “SHIFT + RESET” or “CTRL + RESET” one can stall the emulation and manage it.
- ★ **providing sound output –** using a piezo buzzer, the Apple][emulator can also output original sound effects.
- ★ **future possibility of interfacing an external keyboard –** by adding and reserving some pins on the keyboard controller and carrying them out it is possible to adjust the software later on, in order to let the emulator work with an external keyboard. This could be a PS2 keyboard for example.
- ★ **provision of not only an Apple][original but also an Apple][+ –** as requested in the demand profile at the beginning the only version to implement was an Apple][“original”. Further the Apple][+ was also implemented, featuring the Applesoft BASIC. The user can switch between both in the backend menu.
- ★ **other minor features –** there are some other minor features like the backlight brightness adjustment or reset ability of the emulator, which are not important, but improve other parts of the emulator (backlight adjustment on the durability).

5.1.1 Achieved emulator speed

Most modern emulation or virtualization software needs a fast CPU to run semi modern systems smoothly. Despite the Apple][system is a very basic and old system in comparison to modern systems, they share the same structural problems: the emulation overhead. Hardware implementation is simple. But software implementation of hardware features could become very complex and CPU time consuming. The emulator software must store state variables for the hardware state and keep track of them. This overflow data needs to be managed, which consumes further time. As it turned out in this document the memory mapped I/O is a great problem, because of the many conditionals to check in order to cover every I/O case. A hardware implementation of this part through a simple address decoder is very simple and even faster as it could be in software.

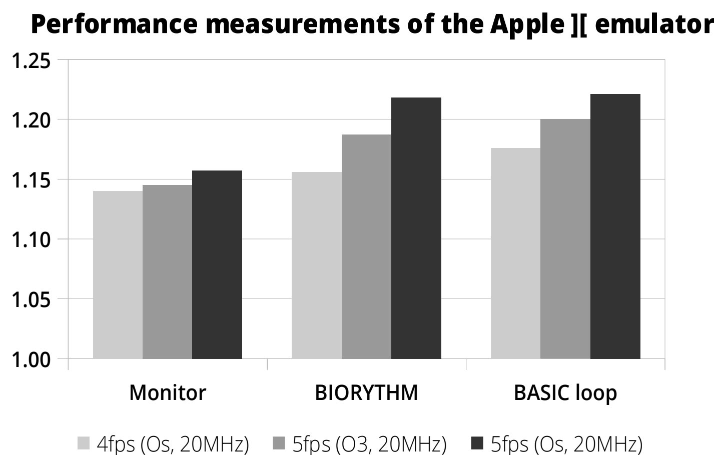


Figure 20: graphical representation of the final performance data.

Figure 20 shows the measured performance data with different screen refresh rates for different use cases. The “Monitor” use case is the simple command line prompt of the Apple][. The “BIORYTHM” is a graphical demonstration program from the DOS 3.3 floppy disk and the BASIC loop is a simple infinite loop, pumping out a string as fast as possible from a BASIC program.

Reducing the frames per second count does not affect the emulation speed seriously, except for the graphical intensive demo program. Also between the very high optimization compiler flag and the standard optimization flag (for program size) is no major difference. Using the “O3” compiler flag will cause an unstable behaviour of the emulator through the very strong optimizations.

Since only a little bit of performance is missing, it has been tried to overclock the ATMega 1284p with a 24 MHz and 25 MHz crystal clock. In both cases the emulator did not even start. So with the clock speed at its limit and the assembler code highly optimized, there is no possibility to improve the performance in the current environment.

The performance of the created emulator is still good as it performs the 1,023,000 clock cycles of the MOS 6502 microprocessor in around 1.19 seconds. This is a variance of only 19% of the original towards the discussed speed problems. The resulting device is totally usable and reacts like a normal Apple][computer.

5.1.2 Unmentioned aspects

This document shows the implementation of the Apple][emulator prototype. But developing this device was a long process. There were other features which did not get into this document but should be mentioned here:

- **breadboard, revision 1 and revision 2 boards** – first of all the design was evaluated and created using a breadboard (figure 21). After this the first prototype of the final prototype was soldered together (revision 1 board) with no external communication abilities (figure 22 and 23). After that, the revision 2 added a serial communication port to download program data from a PC. It also reduced the size of the board in order to be more compact (figure 24, 25 and 26). After this the final prototype (revision 3) was created with a lot of modifications described in this document.

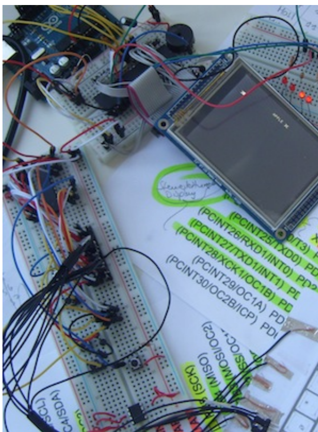


Figure 21: early breadboard version.

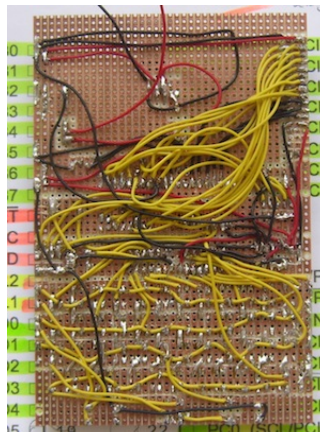


Figure 22: back of the 1st prototype.

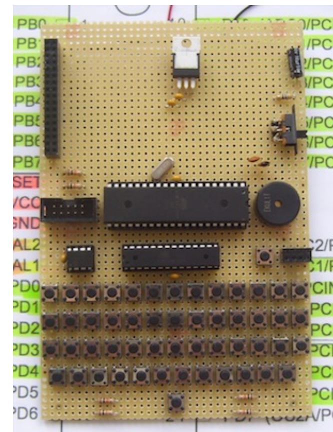


Figure 23: front of the 1st prototype.

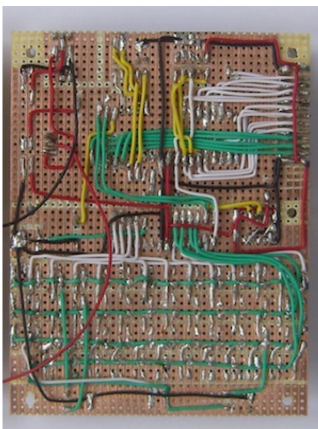


Figure 24: back of the 2nd prototype.

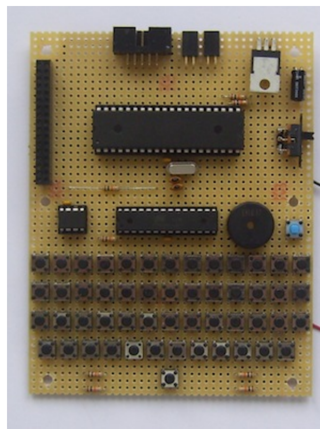


Figure 25: front of the 2nd prototype.

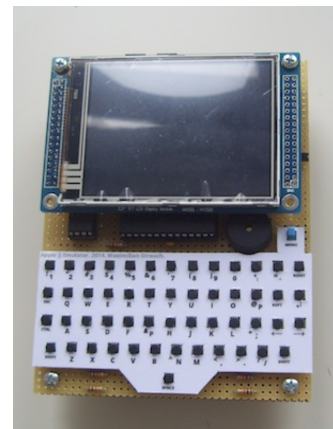


Figure 26: final 2nd prototype.

- **"datalink" for data connection from the PC to the emulator** – the revision 2 board had some connectors to allow the connection of a serial terminal to it. Using an USB-to-UART cable it was possible to connect the emulator to a PC. By a special program written in Java, called "datalink", it was possible to open disk image files and send them to the device. The code of this program was later merged into the revision 3 emulator runtime environment, providing this fea-

ture for files on the SD card from the microcontroller and simplifying this step of loading software.

- **Apple][emulator for the computer** – the really first step of the project was the creation of a computer based version of the Apple][emulator, to get into the details. Since the "home programming language" of the author is Java, it was implemented in Java and later in C. With moving to assembler, the computer version was deprecated due to some bugs (especially in `ADC` and `SBC`).

5.2 Further development

By writing this document and thinking about the future features, which would be nice to have, the author had some ideas for further improvement. Due to time constraints of this thesis it is not possible to implement these new ideas, which are proposed in the following.

The most likely next task is the addition of an external PS2 keyboard, since the hardware USART pins on the keyboard controller are reserved for this purpose and this feature was mentioned some times in the previous chapters.

Additional features, which would be really nice to have are: game strobes, since some games require this type of controller, Disk][drive floppy disk emulation or even the full 48K of main memory enabling more memory for programs and the high resolution mode. These features are simple to implement, but the key problem is heavy performance loss which will degrade the emulated clock cycle speed of the 6502 microprocessor emulation. This loss is caused by the growing length of the memory mapped I/O conditional statements, which need to check for even more cases for these improvements. So a way to replace the memory read macro in the implementation of the 6502 microprocessor emulation needs to be found to add new features.

Since it is written in assembly language, there is no way to speed it up any further. An external static RAM IC could be connected to the microcontroller. It will serve as the physical emulation of the 64K memory of the emulator. Then the ROM is placed in this memory during startup and all memory access can be done very fast, because of a unique interface without conditionals. But there will be two new major problems:

- interfacing an external static RAM IC needs 16 address lines and 8 bidirectional data lines along with at least two control lines. Since the ATMega 1284p has no pins left, the possible options are:
    (1) using the display data bus and some other pins for these purposes. This will result in performance losses, because of the output pins, which are scattered over the entire microcontroller and need to be adjusted on every access with more code.
    (2) using another microcontroller, like the Atmega2560. It increases the 32 pins of the ATMega 1284p to 86 digital I/O pins which would do the job. It is delivered as surface mount component in a smaller SMD package and cannot be soldered easily. But "adapters" are available for this.
- recognition of memory mapped I/O. If a memory mapped I/O address is referenced, like the speaker, a task needs to be performed without writing data or

reading data. So an address decoder or another device, which is very fast and can trigger further actions on special memory addresses on the address bus between emulation microcontroller and the static RAM is needed.

If this obstacle of extending the RAM is taken, it would be very simple to add the other features. It is the barrier for further development.

Chapter 6: Appendix

6.1 Glossary

The following glossary explains some special terms, which are used within this document. Every time a "→" sign is found in front of a word, a basic explanation can be found here.

ASCII – stands for "American Standard Code for Information Interchange" and is a very common and popular 7 bit character-encoding scheme based on the English alphabet with some special characters.

ARM – a RISC microprocessor instruction set architecture (ISA) developed by the British company ARM Holdings. ARM processors are very popular in embedded devices like smartphones, tablet PCs or set-top boxes.

BCD – binary-coded decimal numbers are used to represent fractional numbers accurately. Every digit is represented by a fixed size of bits (mostly four). A BCD number provides a bit block for every digit in the decimal number, generating the highest accuracy on calculation and rounding. For example the decimal number "1.42" is coded as "`0b0001.0100 0010`" (`0x1.42`). This mode was embedded in many early microprocessors (like the MOS 6502) and is nowadays less important, because of IEEE754.

CRT – the "cathode ray tube" is a vacuum tube containing electron guns and a fluorescent screen to display images. In the past CRTs were standard monitor devices until the LCD screen came up. The screen is drawn line by line from left to right and top to bottom. In this context a **HYSNC** event is considered as the time gap when the electronic beam has reached the right end of a line and performs a "carriage return" to the left end (or beginning) of the new line. A **VSYNC** event is then considered as the time gap, when the electronic beam reaches the lower right point of the screen and moves back to the top left point to start drawing a new frame. Generating an output signal needs accurate timing to show an image.

DIP – the "dual in-line package" is a standard electronic device package with a rectangular housing and two parallel rows of I/O pins. The "plastic dual in-line package" (PDIP) specifies the material of the housing of the *die*, which contains the IC.

Dropout voltage – is the smallest difference between the input and output voltage of a voltage regulator. If the input voltage drops bellow the sum of the output voltage and dropout voltage of the regulator, it fails to provide the regulated output voltage.

EEPROM – electrically erasable programmable read-only memory is a type of memory

that is used to store small amounts of data in electronic devices. Since it is non-volatile, the data remains without a steady power supply.

Flash memory – is an electronic, non-volatile memory used to store data. It was developed from EEPROMs in 1984 and is nowadays widely used in SSD hard disks, USB sticks and other components. It is also used as program memory in the Atmel AVR 8 bit microcontroller series.

IC – stands for “integrated circuit” and is considered as a set of electronic components on a semiconducting chip. It has a much smaller size than a discrete circuit made out of single logic gates.

ISA – the “instruction set architecture” specifies the set of operational codes (opcodes) available on a processor and its general architecture, including register files, interrupt handling and other important parts of a processor. There are →RISC and CISC ISAs.

ISP – “in-system programming” is a way to program a microcontroller while it is inside its electronic circuit.

Little endian – a byte encoding with the least significant byte comes first followed by the more significant bytes.

MCU – stands for “microcontroller unit”. See →microprocessor.

Microcontroller – a small “computer” on a single integrated circuit containing a processor, memory and programmable input and output pins.

Microprocessor – also recognized as “CPU” or “processor” is a central processing unit in the housing of an IC without any other hardware. It is only a multi-purpose computation unit. Together with parts like main memory, hard disks, a motherboard, a graphics card and other peripherals a modern personal computer is formed. In contrast to microcontrollers this device can provide more computational power.

Nibble – half of a byte, represented by four bits.

PWM – “pulse width modulation” is a modulation technique to generate a square wave with control over the length of the duty cycle. It allows the control of power passed to an electronic device.

Raspberry Pi – a credit-card-sized single-board computer developed in the UK by the Raspberry Pi Foundation. It is a full featured computer based on the →ARM architecture with the ability to run several Linux desktop and server operating systems for various purposes.

RISC – “reduced instruction set computing” is a CPU design strategy to create a simpler →ISA to archive higher performance through that simplicity. The opposite is a “complex instruction set computing” (CISC) design strategy.

SPI – the “serial peripheral interface” bus is a synchronous serial data transmission standard between electronic devices that operates in full duplex (while sending a byte the sender receives also a byte from the receiver and vice-versa). See page 70 or Atmel application note 151 [46].

SRAM – static random access memory is an integrated circuit that uses flip-flops to store the state “statically”. The opposite is a DRAM, which uses less components, but all

the memory values must be refreshed multiple times a second, which is done by a specific circuitry. This memory is mostly used for memory modules to use in a computer.

System call – a system call is the way how a user program can request resources from the operating system kernel, e.g. access to a file. They provide an essential interface between programs and the operating system kernel to provide security and stability of the system, since the kernel can control the usage of the hardware (e.g. memory protection).

TWI – the two wire interface or I^2C (Inter-Integrated Circuit) is a multi-master computer bus. The data is transmitted serial with the use of only two wires between up to 127 bus participants. See page 69.

U(S)ART – the “universal asynchronous receiver / transmitter” is a communication interface to transmit data serial between two electronic devices on two lines: TXT and RXD. Since it is an asynchronous communication the devices can send and receive at the same time. An USART (“universal synchronous / asynchronous receiver / transmitter”) has also the ability to communicate serially with a clock line. Most Atmel AVR microcontrollers of the ATMega series have a hardware USART interface with the lines RXD and TXD and for synchronous communication also a XCK (clock) line.

UI – abbreviation for “user interface”.

Word – two bytes form a word **–** a 16 bit number, which can represent 65,536 distinct symbols or numbers.

x86 architecture – this is a family of CISC ISAs, based initial on the Intel 8086 CPU, with many extensions, making the architecture very strong and huge. Most modern CPUs use this architecture.

6.2 Bibliography

[1] *Atmel Corporation*, "Fact Sheet", *http://www.atmel.com/about/corporate/factsheet.aspx* (last accessed on 05/30/2014).
[2] *Wikipedia*, "List of home computers", *http://en.wikipedia.org/wiki/List_of_home_computers_by_category* (last accessed on 06/10/2014).
[3] *Jeffrey van der Hoeven, Bram Lohman, Remco Verdegem*, "Emulation for Digital Preservation in Practice: The Results", 2007.
[4] *Stephen A. Edwards*, "Apple2fpga: Reconstructing an Apple II+ on an FPGA", *http://www.cs.columbia.edu/~sedwards/apple2fpga/* (last accessed on 06/10/2014).
[5] *Donald G. Bailey*, "Design for Embedded Image Processing on FPGAs", 2011.
[6] *Cornell University*, "ECE 4760: Designing with Microcontrollers", *http://people.ece.cornell.edu/land/courses/ece4760/* (last accessed on 06/10/2014).
[7] *Cornell University*, "Apple II Emulator", *https://instruct1.cit.cornell.edu/courses/ee476/FinalProjects/s2007/bcr22/final%20webpage/final.html* (last accessed on 06/10/2014).
[8] *Tom Gowing, Brian Pescatore*, "NES Emulation", *http://people.ece.cornell.edu/land/courses/ece4760/FinalProjects/s2009/bhp7_teg25/bhp7_teg25/index.html* (last accessed on 06/10/2014).
[9] *mikrocontroller.net*, "AVR 6502 Emulator", *http://www.mikrocontroller.net/articles/AVR_6502_Emulator* (last accessed on 06/10/2014).
[10] *Charles Ingerham Peddle, Wilbur L. Mathys, William D. Mensch, Jr., Rodney H. Orgill*, "Integrated circuit microprocessor with parallel binary adder having on-the-fly correction to provide decimal results (US 3991307 A)", 1976, *http://www.google.com/patents/US3991307*.
[11] *Wikipedia*, "Term of patent in the United States", *http://en.wikipedia.org/wiki/Term_of_patent_in_the_United_States* (last accessed on 06/10/2014).
[12] *Computer History Museum*, "Apple II DOS source code", *http://www.computerhistory.org/atchm/apple-ii-dos-source-code/* (last accessed on 06/10/2014).
[13] *Charlotte Erdmann*, ""One more thing": Apples Erfolgsgeschichte vom Apple I bis zum iPad", 2011.
[14] *Wikipedia*, "MOS Technology 6502", *http://en.wikipedia.org/wiki/MOS_Technology_6502* (last accessed on 06/10/2014).
[15] *MOS Technology, Inc.*, "MCS6500 Microprocessors, The MCS6500 Microprocessor Family Concept (datasheet)", 1976.
[16] *Intel Corp.*, "8-Bit N-Channel Microprocessor (datasheet)", 1974.
[17] *MOS Technology, Inc.*, "MOS Microcomputers, Programming Manual (MCS6500)", 1976.
[18] *MOS Technology, Inc.*, "MOS Microcomputers, Hardware Manual", 1976.
[19] *visual6502.org*, "Visual Transistor-level Simulation of the 6502 CPU", *http://www.visual6502.org/* (last accessed on 06/10/2014).
[20] *Michael Steil, Melissa, Sebastian Biallas*, "Internals of BRK/IRQ/NMI/RESET on a MOS 6502", *http://www.pagetable.com/?p=410* (last accessed on 06/10/2014).
[21] *Christopher Espinosa*, "Apple][Reference Manual", 1979.
[22] *James F. Sather*, "Understanding the Apple II", 1983.
[23] *Wikipedia*, "List of 8-bit computer hardware palettes", *en.wikipedia.org/wiki/List_of_8-bit_computer_hardware_palettes#Apple_II_series* (last accessed on 06/10/2014).
[24] *Dave Schmenk*, "Apple II Pi adapter card", *http://schmenk.is-a-geek.com/wordpress/?p=167* (last accessed on 06/10/2014).
[25] *Apple Computer Inc.*, "Apple II Reference Manual, "Redbook"", 1978.
[26] *D. A. Godse, A. P. Godse*, "Microprocessor, Microcontroller & Applications", 2008.
[27] *John Crisp*, "Introduction to Microprocessors and Microcontrollers", 2003.
[28] *Rolf Gübeli*, "Technische Informatik: Mikroprozessor-Hardware und Programmiertechniken (Band 2)", 2004.
[29] *Wikipedia*, "Von Neumann architecture", *http://en.wikipedia.org/wiki/Von_Neumann_architecture* (last accessed on 06/11/2014).

[30] *Wikipedia*, "Harvard architecture", *http://en.wikipedia.org/wiki/Harvard_architecture* (last accessed on).

[31] *Wikipedia*, "Modified Harvard architecture", *http://en.wikipedia.org/wiki/Modified_Harvard_architecture* (last accessed on 06/11/2014).

[32] *John Pickens, Bruce Clark, Ed Spittles* , "NMOS 6502 Opcodes", *http://www.6502.org/tutorials/6502opcodes.html* (last accessed on 06/11/2014).

[33] *Free Software Foundation, Inc.*, "A GNU Manual, 6.3 Labels as Values", *http://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html* (last accessed on 06/11/2014).

[34] *Atmel Corp.*, "Atmel 8-bit AVR Instruction Set", 2010.

[35] *Wikipedia*, "Apple II", *http://en.wikipedia.org/wiki/Apple_II* (last accessed on 06/10/2014).

[36] *Wikipedia*, "Apple II+", *http://en.wikipedia.org/wiki/Apple_II%2B* (last accessed on 06/10/2014).

[37] *Elm Chan*, "Petit FAT File System Module", *http://elm-chan.org/fsw/ff/00index_p.html* (last accessed on 06/11/2014).

[38] *Apple Computer Inc.*, "Applesoft][, BASIC Programming Reference Manual", 1978, 1981.

[39] *Atmel Corp.*, "8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash (datasheet ATMega1284p)", 2011.

[40] *Atmel Corp.*, "Atmel AVR042: AVR Hardware Design Considerations", 2013.

[41] *Solomon Systech Limited*, "SSD1289, Advance Information, 240 RGB x 320 TFT LCD Controller Driver integrated Power Circuit, Gate and Source Driver with built-in RAM (datasheet) ", 2007.

[42] *Microchip Technology Inc.*, "1024K I2C CMOS Serial EEPROM, 24AA1025 / 24LC1025 / 24FC1025 (datasheet)", 2005.

[43] *Alexander Starke*, "AVR TWI", *http://www.mikrocontroller.net/articles/AVR_TWI* (last accessed on 06/12/2014).

[44] *Atmel Corp.*, "AVR311: Using the TWI module as I2C slave", 2009.

[45] *Atmel Corp.*, "AVR910: In-System Programming", 2008.

[46] *Atmel Corp.*, "AVR151: Setup And Use of The SPI", 2008.

[47] *Merten Joost*, "Mikrocontroller und Robotik: von der CPU zum Computer", 2013.

[48] *Atmel Corp.*, "8-bit Atmel with 8KBytes In-System Programmable Flash (datasheet ATMega8)", 2013.

6.3 "Speed" measurement setup

To measure the speed of a certain operation on the microcontroller one can implement a "stopwatch" on the microcontroller. This will not give accurate results, because the microcontroller, which should be measured, measures itself and distorts the result.

A better approach is to set an output pin before the operation, which should be measured, to logical high. If the operation is finished the pin is pulled back to logical low. Connecting this output pin to an Arduino Uno and loading the sketch from listing 19 onto the Uno creates a simple "stopwatch" setup. Since the output pin of the microcontroller is logical high during the operation, the Uno can measure the duration this pin remains high without distorting the microcontroller, which is measured, and print the time on a serial terminal.

The sketch uses an Arduino pin as input which triggers an interrupt when the logical level at the pin changes. With some if–conditions and a serial connection to the computer, the result data can be streamed very easily to the computer and picked from there. The time inaccuracy of this method is far less than 1 ms, which is enough for the purposes used in this document.

```
1 // Time watch to watch the state of a digital pin.
  // @see http://arduino.cc/de/Reference/AttachInterrupt
  unsigned long volatile duration;
  #define PIN 3
5
  void setup() {
    // Serial connection
    Serial.begin(9600);
    Serial.println("Time watch. 2014. Maximilian Strauch.");
10  // Set up port
    pinMode(PIN, INPUT);
    attachInterrupt(1, tick, CHANGE);
    duration = 0;
  }
15
  void loop() {
    // Nothing to do here
  }

20  void tick() {
    if (duration == 0) {
      // Save start time
      duration = micros();
    } else {
25    // One cylce: output time and reset
      duration = micros() - duration;
      Serial.println(duration);
      duration = 0;
    }
30  }
```

Listing 19: Arduino Uno sketch to measure the time an input pin is pulled high.