A Motorola 68000 Emulator for Palm OS running on Android

Computer Science Tripos – Part II
Pembroke College

Post-submission version
with corrections
October 5, 2016
Corrections

The following minor errors were corrected post-submission. Please do let me know if you find any more.

Proforma

page i  Corrected wordcount.
page i  “MC68VZ326” corrected to “MC68VZ328”.

Implementation

pages 14, 16  Fixed broken references to tables.
page 33  Replaced mismatched bracket with dash.

Evaluation

page 38  Added missing article (paragraph 3).
Proforma

Name: Seb Aebischer
College: Pembroke College
Project Title: A Motorola 68000 Emulator for Palm OS running on Android
Examination: Computer Science Tripos Part II, 2015-16
Word Count: 11800 approx.
Project Originator: Seb Aebischer
Project Supervisor: Ripduman Sohan

Original aims of the project

To emulate an abstract 68k processor; to emulate the MC68VZ328 processor; to emulate, to the greatest extent permitted in the project timeframe, the Palm m515 handheld computer.

Summary of work completed

A core 68k emulator was implemented, and compiled into both a Windows executable and Android app. A MC68VZ328 emulator was implemented using this as a starting point. This was sufficient to meet the main goals of the project. Additionally extension work was done to enable Palm OS to be successfully booted and interacted with using the smartphone’s touchscreen, as well as a number of tools to assist with this. The work was evaluated quantitatively and qualitatively through a number of benchmark programs and extensive testing.

Special difficulties

None.
Declaration of originality

I, Seb Aebischer of Pembroke College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date
## Contents

1 Introduction ................................. 1
   1.1 Emulators ................................ 1
   1.2 Motorola 68000 .......................... 2
   1.3 Related Work .............................. 2
   1.4 Development Environment ................ 2

2 Preparation ................................. 3
   2.1 The 68000 Architecture .................. 3
      2.1.1 Instruction Format ................... 4
      2.1.2 Exception Processing ................. 6
   2.2 The MC68VZ328 DragonBall ............... 7
      2.2.1 On-chip Peripherals .................. 7
      2.2.2 Chip Select .......................... 7
      2.2.3 Interrupt Controller ................. 8
      2.2.4 Serial Peripheral Interfaces ....... 8
   2.3 The Palm m515 ............................ 9
      2.3.1 Memory ............................... 9
      2.3.2 Epson S1D13706 LCD controller ....... 9
      2.3.3 BurrBrown ADS7846E analogue-digital converter (ADC) .... 9
      2.3.4 Philips PDIUSBD12 USB interface .... 10
   2.4 New languages and skills ............... 10

3 Implementation .............................. 11
   3.1 Project Structure ........................ 11
   3.2 Disassembling 68k Machine Code .......... 12
      3.2.1 Overview ............................ 12
      3.2.2 Typedefs ............................ 12
      3.2.3 Identifying Instructions .......... 13
      3.2.4 Pretty-printing .................... 13
   3.3 Turning it into an Emulator ............ 14
      3.3.1 Register File and Internal State .... 14
      3.3.2 Decoding Effective Addresses ...... 15
      3.3.3 Conditional Operations ............. 16
      3.3.4 Fetching and Executing Instructions 16
      3.3.5 Exceptions and Interrupts ........... 17
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Structure of the 68k status register. Bits 0-7 form the CCR, and bits 8-15</td>
<td>3</td>
</tr>
<tr>
<td>2.2</td>
<td>Common fields in the binary representation of 68k instructions.</td>
<td>4</td>
</tr>
<tr>
<td>2.3</td>
<td>Illustration of the master-slave serial peripheral interface.</td>
<td>8</td>
</tr>
<tr>
<td>2.4</td>
<td>Timing diagram of SPI communication with the ADS7846E. There is a one clock cycle delay between the end of the command byte and the ADC's response.</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>The development process resembled the waterfall model.</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>Class diagram for the DragonBall emulator. The DragonBallPeripheral subclasses are not all shown.</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>Capture of the Palm OS boot screen, created using the display controller's exportToBitmap method.</td>
<td>30</td>
</tr>
<tr>
<td>3.4</td>
<td>Screenshots of the Android app created for the project. The left shows the session list, and the right shows the emulator running.</td>
<td>32</td>
</tr>
<tr>
<td>3.5</td>
<td>Old and new, side-by-side: the Palm m515 used in this project next to its emulated version on an Android smartphone.</td>
<td>34</td>
</tr>
<tr>
<td>4.1</td>
<td>Average execution time of each benchmark program over three runs, in five test environments.</td>
<td>37</td>
</tr>
<tr>
<td>A.1</td>
<td>Screenshot of the Palm benchmark-runner application, m68test.</td>
<td>46</td>
</tr>
</tbody>
</table>
List of Tables

2.1 The subset of 68k addressing modes present in the MC68VZ328. The Encoding column shows how each is represented in the 6-bit effective address field. *** and ** indicate information that is stored in extension words. Quick immediates and implied registers are used only by certain instructions. Source: Table 1-1 in the *MC68VZ328 User’s Manual* [3] . . 5

2.2 Condition codes and their corresponding binary encoding and predicates. Source: Table 3-19 in the *MC68000 Programmer’s Reference Manual* [2] 5

4.1 Relative time taken to execute a register and memory operation in the on each platform. The ratio shows how many times slower a memory operation was than a register operation. The Palm emulator results were used for the Windows and Android columns. . . . . . . . . . . . . . . . . 38

A.1 Summary of the benchmark programs. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

B.1 Average execution time in milliseconds measured over three runs of each benchmark program run, in the five test environments. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
Chapter 1

Introduction

Back in 2000, a company known as Palm was leading the market of personal digital assistants (PDAs) with its innovative user-centred software and high-quality handheld devices. The m515, released in 2002, had reasonable specifications for its day: a colour 160-by-160 pixel touchscreen, 16 MB of memory, and a 33 MHz processor.

Fast-forward 14 years. Palm no longer exists, what were once cutting-edge devices are either discarded or gathering dust in attics. Mobile computing has improved phenomenally thanks to Moore’s law: a typical modern smartphone features gigabytes of memory and a multi-core processor clocked at over 1 GHz, not to mention high-resolution screens and dozens of embedded sensors.

My objective in this dissertation project was to produce an emulator capable of emulating the Palm m515. As an extra challenge, the emulator was designed so that it could be run on both my laptop and an Android smartphone. My motivation for this is twofold: writing an emulator is a good way to learn about a hardware system, which interests me; emulators are also a means of digitally preserving obsolete hardware, and my physical m515 is nearing the end of its life.

1.1 Emulators

The focus of this project is emulation. In context, this means programming one computer system to mimic another, allowing machine code compiled for the latter platform (the guest) to be run on the former (the host). When emulating a processor, one of two approaches is usually employed: instructions are interpreted, or they are recompiled to the host’s native machine code.

Recompilation is a wide and interesting field, subcategorised into static and dynamic ‘just-in-time’ (JIT) recompilation. The latter is preferred because it allows emulation of self-modifying code. While JIT is substantially faster than an interpreter, it was not considered for this project for three reasons. Firstly, a dynamic recompiler is the more complex of the two, and would probably make a large Part II project by itself. Secondly, dynamic code execution is non-trivial on an Android device. Most Android apps run inside a Java Virtual Machine (JVM), which abstracts away the underlying hardware, be it ARM or another platform. Native machine code would have to be generated differently
for each platform, and while I could focus on only one, it does not result in a general cross-platform solution. Lastly, modern processors are so much faster than those of 14 years ago, not only as a result of higher clock frequency but also improved pipelining and out-of-order execution technologies, that JIT should not be necessary to achieve a reasonable emulation speed of something a couple of orders of magnitude slower.

1.2 Motorola 68000

At the heart of the m515 is the MC68VZ328 “DragonBall” processor [3], the core of which is based on Motorola’s 68000 architecture. Colloquially referred to as 68k, this is a CISC architecture that featured in a number of well-known computers, including the original Apple Macintosh and Commodore Amiga.

While it follows the fairly standard register machine model, in this case providing a total of 16 registers and instructions that operate on and move data between them, it was unusual in that it was designed with ease of programming in mind: most operand modes could be used with most instructions, and several instructions were added to assist with high-level programming constructs like subroutines.

Details of the architecture relevant to this project are covered in the next chapter.

1.3 Related Work

The Palm software development kit (SDK) included the official Palm OS Emulator (POSE), which included a number of useful development tools such as memory leak detection and support for external debuggers. POSE was based on Copilot [8], an emulator for some of the original Palm Pilot devices, written by Greg Hewgill. Copilot was itself based on UAE, an emulator for the Commodore Amiga. POSE provided the inspiration for this project.

After starting the initial research for the project I came across PHEM, the Palm Hardware Emulator [7], which is even closer to what I hoped to achieve since it runs on Android – though compatible with ARM-based devices only. PHEM is a port of POSE.

1.4 Development Environment

Almost all the work in this project was carried out on my laptop (Windows 64-bit, Intel Core i7, 2.1 GHz). My desktop computer and the lab machines were available as a fallback. Testing of the emulator took place on both my laptop and my Android smartphone (Android 4.4, Samsung Exynos 5260, 1.3 GHz).

The emulator was written in C/C++, using an IDE of my own invention, and compiled using MinGW GCC 4.9.3. Java was also used when developing the Android app to create the user interface and related threads; the emulator routines were accessed via the Java Native Interface (JNI). App development was done using Android Studio.
Chapter 2

Preparation

This project demanded a lot of information in many different areas. In this chapter I shall cover the pertinent areas I researched, and present summaries of my findings. First an introduction to the 68k architecture is given. This is followed by details of the Palm hardware, including the 68k-based processor and its on-chip peripherals, and other components found in the Palm that were important during this project.

2.1 The 68000 Architecture

As mentioned earlier, the 68k is register-based, providing a total of 16 32-bit registers. These are divided into two groups: 8 data registers (named D0–D7) and 8 address registers (named A0–A7). There is also a program counter (PC), and a 16-bit status register (SR), which is split into a protected system byte and a user byte known as the condition code register (CCR). The structure of the status register is shown in Figure 2.1. The CCR is updated according to the result of operating on data registers. Address registers behave differently: operations on them do not affect the CCR.

The address register A7 is used as the program’s stack pointer. There are in fact two stack pointers, a user stack pointer (USP) and a supervisor stack pointer (SSP). This allows user and kernel stacks to be maintained separately, which is important for security and thread-safety on multi-tasking operating systems. Which stack pointer is currently active is determined by the supervisor (S) bit in SR; any operation on A7 will operate on only the active stack pointer.

![Figure 2.1](image-url)

**Figure 2.1:** Structure of the 68k status register. Bits 0-7 form the CCR, and bits 8-15 form the system byte. Greyed bits are unused.
There are three primary data sizes: bytes (8 bits), words (16 bits), and longs (32 bits). Most data-related instructions are capable of operating on all sizes. Additionally there are special instructions for dealing with single bits and binary-coded decimal (BCD) numbers. The processor’s data bus is 16 bits wide, and the address bus is 32 bits wide. It is however compatible with peripherals that work over an 8-bit data bus. In this case only the lower byte of the data bus is used, and the access address of the device is half the address on the address bus.

### 2.1.1 Instruction Format

All instructions comprise one or more consecutive words in memory. The first word is the instruction word, which defines which operation will be performed. This is optionally followed by a number of extension words and data words, which can provide additional information such as immediate data or address offsets.

The binary format of instruction words is consistent between similar instructions. Some examples are given in Figure 2.2.

#### Typical instruction operating on effective address and register (e.g. ADD)

<table>
<thead>
<tr>
<th>TARGET REGISTER</th>
<th>SIZE</th>
<th>EFFECTIVE ADDRESS MODE</th>
<th>REGISTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 14 13 12 11 10 9 8</td>
<td>7 6</td>
<td>5 4 3 2 1 0</td>
<td></td>
</tr>
</tbody>
</table>

#### Instruction operating on quick immediate (e.g. MOVEQ)

<table>
<thead>
<tr>
<th>QUICK IMMEDIATE</th>
<th>SIZE</th>
<th>EFFECTIVE ADDRESS MODE</th>
<th>REGISTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 14 13 12 11 10 9 8</td>
<td>7 6</td>
<td>5 4 3 2 1 0</td>
<td></td>
</tr>
</tbody>
</table>

#### Conditional instructions (e.g. Bcc)

<table>
<thead>
<tr>
<th>CONDITION CODE</th>
<th>SIZE</th>
<th>EFFECTIVE ADDRESS MODE</th>
<th>REGISTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 14 13 12 11 10 9 8</td>
<td>7 6</td>
<td>5 4 3 2 1 0</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2.2:** Common fields in the binary representation of 68k instructions.

The effective address field allows a great many addressing modes, and the architecture was designed so that they are all consistent, and most of them can be used with most instructions. While the exact number of modes varies with the processor version, the MC68VZ328 (see Section 2.2 below) has no fewer than 14 different ways to specify an operand’s effective address, which the User’s Manual divides into 6 categories: register direct, register indirect, absolute, program counter relative, immediate, and implied.

The addressing modes are summarised in Table 2.1. In some cases, indicated by xxx, additional information is required to express the address; this is what extension words are for. For instructions like MOVE that have two effective addresses, the source operand’s extension words come first.

The instruction’s size field determines the data type it operates on. When an instruction operates on a data register as a byte or word, the higher bits remain unchanged; thus
### Table 2.1: The subset of 68k addressing modes present in the MC68VZ328. The Encoding column shows how each is represented in the 6-bit effective address field. \( xxx \) and \( XX \) indicate information that is stored in extension words. Quick immediates and implied registers are used only by certain instructions. Source: Table 1-1 in the *MC68VZ328 User’s Manual* [3]

<table>
<thead>
<tr>
<th>Mode</th>
<th>Encoding</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data register</td>
<td>000 nnn</td>
<td>Dn</td>
</tr>
<tr>
<td>Address register</td>
<td>001 nnn</td>
<td>An</td>
</tr>
<tr>
<td>Register indirect</td>
<td>010 nnn</td>
<td>((An))</td>
</tr>
<tr>
<td>Register indirect with postincrement</td>
<td>011 nnn</td>
<td>((An))+</td>
</tr>
<tr>
<td>Register indirect with predecrement</td>
<td>100 nnn</td>
<td>-(An)</td>
</tr>
<tr>
<td>Register indirect with 16-bit offset</td>
<td>101 nnn</td>
<td>(xxx(An))</td>
</tr>
<tr>
<td>Indexed register indirect with 8-bit offset</td>
<td>110 nnn</td>
<td>(xxx(An, XX))</td>
</tr>
<tr>
<td>16-bit absolute</td>
<td>111 000</td>
<td>(xxx.w)</td>
</tr>
<tr>
<td>32-bit absolute</td>
<td>111 001</td>
<td>(xxx.l)</td>
</tr>
<tr>
<td>PC-relative with offset</td>
<td>111 010</td>
<td>(xxx(An))</td>
</tr>
<tr>
<td>Indexed PC-relative with offset</td>
<td>111 011</td>
<td>(xxx(An))</td>
</tr>
<tr>
<td>Immediate</td>
<td>111 100</td>
<td>(#xxx)</td>
</tr>
<tr>
<td>Quick (3-bit) immediate</td>
<td>—</td>
<td>#1 to #3</td>
</tr>
<tr>
<td>Implied register</td>
<td>—</td>
<td>SR/USP/SP/PC</td>
</tr>
</tbody>
</table>

### Table 2.2: Condition codes and their corresponding binary encoding and predicates. Source: Table 3-19 in the *MC68000 Programmer’s Reference Manual* [2]

<table>
<thead>
<tr>
<th>Condition</th>
<th>Encoding</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>0000</td>
<td>T</td>
</tr>
<tr>
<td>False</td>
<td>0001</td>
<td>F</td>
</tr>
<tr>
<td>High</td>
<td>0010</td>
<td>(C \land Z)</td>
</tr>
<tr>
<td>Low or Same</td>
<td>0011</td>
<td>(C \lor Z)</td>
</tr>
<tr>
<td>Carry clear</td>
<td>0100</td>
<td>C</td>
</tr>
<tr>
<td>Carry set</td>
<td>0101</td>
<td>C</td>
</tr>
<tr>
<td>Not equal</td>
<td>0110</td>
<td>Z</td>
</tr>
<tr>
<td>Equal</td>
<td>0111</td>
<td>Z</td>
</tr>
<tr>
<td>Overflow clear</td>
<td>1000</td>
<td>(\lnot)</td>
</tr>
<tr>
<td>Overflow set</td>
<td>1001</td>
<td>(V)</td>
</tr>
<tr>
<td>Plus</td>
<td>1010</td>
<td>(N)</td>
</tr>
<tr>
<td>Minus</td>
<td>1011</td>
<td>(N)</td>
</tr>
<tr>
<td>Greater or equal</td>
<td>1100</td>
<td>((N \land V) \lor (\lnot N \land \lnot V))</td>
</tr>
<tr>
<td>Less than</td>
<td>1101</td>
<td>((N \land V) \lor (\lnot N \land V))</td>
</tr>
<tr>
<td>Greater than</td>
<td>1110</td>
<td>((N \land V \land Z) \lor (N \land \lnot V \land Z))</td>
</tr>
<tr>
<td>Less or equal</td>
<td>1111</td>
<td>(Z \lor (N \land \lnot V) \lor (N \land V))</td>
</tr>
</tbody>
</table>
writing $8765.w$ to a register containing $12345678$ will result in $12348765$. Address operations do not support the byte data size. When an address is operated on as a word, the actual operation takes place with a 32-bit sign-extended copy of the word. Thus writing $8765.w$ to an address register containing $12345678$ will result in $ffff8765$.

When the immediate mode is used, the data size and therefore number of extension words is determined by the instruction’s size. Quick immediates are used only by the ADDQ, SUBQ, and MOVEQ instructions and form a 3-bit field in its opcode. Implied registers are also handled only by special instructions.

### 2.1.2 Exception Processing

Exceptions provide a way of handling events that interrupt the normal flow of program execution. In this way they are similar to the usual concept of CPU interrupts. However, interrupts make up a subset of exceptions; an exception may also be caused internally by certain conditions or instructions, such as privilege violation or the TRAP instruction. Each exception has an exception vector number, from 0 to 255, that identifies its cause and is used to select the appropriate handler routine.

An exception is processed according to the following procedure, taken from section 6.2 of the *M68000 User’s Manual* [1]. I have added some extra information for each step.

1. Make a temporary copy of the status register and set the status register for exception processing.

   The status register is updated by by setting the supervisor (S) bit and clearing the trace (T) bits, which puts the processor into supervisor mode and turns off tracing.

2. Obtain the exception vector.

   The exception vector table is stored in the first 1024 bytes of the address space and comprises 256 32-bit addresses. The exception vector number is left-shifted two bits to multiply it by four, and the result is then used as the address to read the exception vector from.

3. Save the current processor context.

   Most of the time this simply involves pushing the old status register and the program counter to the stack. Other cases were not considered in the project.

4. Obtain a new context and resume instruction processing.

   Most of the time this involves setting the program counter to the exception vector. Again, other cases were not considered in the project.

   There is a special-case exception that occurs when the processor is reset, which occurs when it is powered on or experiences a catastrophic failure. Steps 1 to 3 are replaced by the two steps described below.

1. Set the status register.

   No copy of the status register is saved. The supervisor (S) bit is set, the trace (T) bits are cleared, and the interrupt priority level (I) is set to 7.
2. Obtain the initial stack pointer and reset vector.

Unlike other exceptions, there are two vectors associated with it in the exception
vector table, occupying positions 0 and 1 (addresses $0000$ and $0004$). The first is
the initial stack pointer, and is loaded into SSP. The second is the initial program
counter, and is loaded into PC.

2.2 The MC68VZ328 DragonBall

The DragonBall is centred around the FLX68000 core, a lower-power variant of 68000
that has a full 32-bit address bus, but lacks a memory management unit (MMU) and
several groups of instructions, such as bitfield manipulation and coprocessor commands.
The MC68VZ328 User’s Manual [3] provided all the necessary information about the
chip, although in places it was unclear and contained one or two mistakes. These areas
were clarified experimentally or through trial-and-error.

2.2.1 On-chip Peripherals

The DragonBall reserves a 4 KB region of the address space for internal registers that
control a multitude of on-chip peripherals. This region covers all addresses that match
$\text{xxxx}$, where x is a don’t-care nibble, or optionally $\text{xxxx}$ if a particular control
bit is set in one of these registers. The region is subdivided into 256-byte blocks in which
registers for a peripheral module, or in some cases more than one, is mapped. Throughout
the rest of this document I shall regard any modules that share the same 256-byte block
as one peripheral.

Only the peripherals necessary for running Palm OS were implemented in the project
in order to fit time constraints. Details of the most interesting ones that required a
significant degree of understanding are given in the proceeding subsections.

2.2.2 Chip Select

There are 8 general-purpose chip-select signals, CSA0 and CSA1 through to CSD0 and
CSD1. Each group has a base address register and control register that includes an enable
bit and size field, plus some group-specific flags. Two requirements are imposed: the base
address must be aligned on a multiple of the chip-select size, and no two chip-select ranges
should overlap. On reset, all chip-selects are disabled, and in this state CSA0 is always
asserted. This remains the case until one of the chip-selects is enabled.

When a chip-select is enabled, it is asserted whenever an address falls in the range
given by the base address base and the chip-select size size. It is not explicitly stated in
the documentation how the two chip-select in each group are mapped. It was decided
that they cover two consecutive ranges: CSx0 ranges from base to base+size, and CSx1
ranges from base+size to base+2×size.

The chip-select’s size field is 3 bits wide, allowing a choice from 8 pre-defined sizes
ranging from 128 KB to 16 MB for CSA and CSB, and 32 KB to 4 MB for CSC and
CSD. There is an additional bit called DSIZ3 in the chip-select control register, which
according to the documentation can increase the range of CSC and CSD up to 16 MB. It was found that in practice it does not affect CSC. This was deduced by inspection of registers as set up in the Palm m515: CSC is configured with base address $10400000 with size bits 000, which corresponds to 8 MB with DSIZ3 set. However, $10400000 is not aligned on an 8 MB boundary, so this would be an invalid configuration.

2.2.3 Interrupt Controller

The 68k core has a very simple interface for receiving external interrupts, in the form of a 3-bit interrupt level input. Zero indicates no interrupt, and seven indicates the highest priority interrupt. It is the role of the interrupt controller to receive interrupts and pass the highest priority to the core.

Among the interrupt controller’s registers are the interrupt status register (ISR), pending register (IPR), and mask register (IMR). The IPR mirrors the state of external interrupt lines: if an interrupt is asserted, its corresponding bit is 1, and if it is deasserted, it is 0. Nothing further happens if the interrupt is masked in the IMR. Only if the corresponding IMR bit is 0 is the bit set in the ISR. The controller reports the highest priority of interrupts whose bits are set in the ISR.

The processor checks for interrupts at the end of an instruction cycle. If the interrupt level is non-zero, it is compared to the IPL in the status register. The interrupt is acknowledged only if its level is greater than the IPL, unless it is a level 7 interrupt. On acknowledgement, the interrupt controller places an interrupt vector number on the data bus, and the processor performs an exception with that number. The number is determined by adding the interrupt level to the value in the interrupt vector register (IVR). Once the interrupt has been processed, the IPL is updated to the interrupt level, which prevents re-triggering after the next instruction cycle.

2.2.4 Serial Peripheral Interfaces

The DragonBall provides two serial peripheral interfaces (SPI). The interface consists of three wires, as shown in Figure 2.3. SPI operates by exchanging bits over DIN and DOUT between master and slave devices on each cycle of DCLK.

![Figure 2.3: Illustration of the master-slave serial peripheral interface.](image-url)
2.3 The Palm m515

Apart from the processor, there were five other hardware components of the m515 that it was necessary to research in order to create a usable emulator. While these are not, to my knowledge, formally documented, I had access to a disassembled m515 and was able to identify the components on the motherboard. Their interfaces to the MC68VZ328 were determined by reverse engineering.

2.3.1 Memory

There are two memory chips present: 4 MB Flash that stores the ROM image, and 16 MB SDRAM that provides user data storage. It was determined that these are connected to chip-select signals CSA and CSD respectively, by inspecting the chip-select register configuration on the m515.

2.3.2 Epson S1D13706 LCD controller

This component drives the m515’s display and provides 80 KB of on-board memory, with a large number of memory-mapped control registers. It was determined that this is connected to the CSB chip-select signal, again by inspection of the physical device. The manual also helpfully contained a section on interfacing with the DragonBall VZ.

The following information was obtained from the *S1D13706 Technical Manual*. The S1D13706 supports up to 16-bit colour, with lower colour depths making use of a lookup table (LUT). The m515 uses 8-bit colour, so it was necessary to understand how this works. The LUT is programmed by writing each RGB component to the three Look-up Table Write Data registers, followed by writing the colour index to the Look-up Table Write Address register.

It was also necessary to get certain display parameters from the registers. For this most part this is straightforward, when the desired values are encoded directly in the registers. For example, the selected colour depth is found in bits 0–2 of the Display Mode register. Because each register is only a byte in size, some longer values are stored in multiple registers. The start address for the screen image can be found in the PIP+ Display Start Address registers.

Others parameters require some calculations. The manual provides equations 2.1 and 2.2 for finding the width and height of the display in pixels.

\[
\text{Display width} = (\text{Horizontal display period} + 1) \times 8 \tag{2.1}
\]
\[
\text{Display height} = (\text{Vertical display period} + 1) \tag{2.2}
\]

2.3.3 BurrBrown ADS7846E analogue-digital converter (ADC)

This component is essentially a multi-input ADC, and provides the interface to the touchscreen digitiser. It connects to the processor via SPI, and by inspecting the boot code it was determined that it is connected to SPI2 of the DragonBall.
The information presented here was obtained from the *ADS7846 Datasheet* [[6]]. Commands are sent to the ADC by exchanging 8 command bits over SPI. This first bit it always 1. This is followed by a 3-bit channel number, which selects the input for measurement by the ADC, and a mode bit, which selects the conversion precision – 8 or 12 bits. The remaining three command bits comprise a further mode bit, and two power control bits. Neither of these are particularly important in the scope of the project, except that if both power control bits are 0, the ADC is turned off and does not proceed with conversion.

Four data clock cycles after measurement acquisition begins – one clock cycle after the last command bit – the digitised value is read out MSB-first by exchanging the appropriate number of bits. Figure 2.4 shows a timing diagram for a 12-bit conversion.

### 2.3.4 Philips PDIUSBD12 USB interface

The last component of importance deals with the m515’s USB communications, and is connected to the CSC chip-select signal, as determined by register inspection. The *PDIUSBD12 Product Data Sheet* [[4]] supplied details of its operation. Unlike the other hardware, the PDIUSBD12 is an 8-bit device.

Communication takes place through a series of transactions, which are summarised in section 10 of the datasheet. A transaction is initiated by writing a command byte to the command register. Depending on the command, a certain number of reads or writes are made to the data register. When this has been completed the transaction is implicitly ended.

### 2.4 New languages and skills

Doing this project required learning an assembly language, which was done with the help of numerous online tutorials. The majority of the project was written in C++, in which I had a little experience during the Part IB course but had not used since. I was however very familiar with C, so picking it up again presented no significant difficulties.

Use of the Android Native Development Kit (NDK) and Java Native Interface (JNI) was required when developing Android apps to call the C++ emulator routines from the main Java code. Additionally getting NDK projects to build in Android Studio required modification of several Gradle configuration files, but this process was well-documented on the Android Developer website.

---

**Figure 2.4:** Timing diagram of SPI communication with the ADS7846E. There is a one clock cycle delay between the end of the command byte and the ADC’s response.
Chapter 3

Implementation

In the previous chapter I went into some detail about the 68k architecture and Palm m515 hardware. This chapter documents the work undertaken to create a working emulator from this information.

3.1 Project Structure

The project implementation was split into three distinct phases. First, a disassembler was written that takes compiled 68k code and prints out the corresponding assembly instructions. This was then used as the starting point for a 68k core emulator, which decodes and executes instructions on a virtual set of registers and memory. Finally, the Palm emulator was built around this core by expanding the memory interface to support the many peripheral devices found in the DragonBall, and providing software implementations of these devices.

The series of phases resulted in a waterfall-style approach to the project, shown in Figure 3.1. Each phase relied on the preceding ones, and often it was necessary to revisit earlier phases when a bug in the emulator was discovered.

Figure 3.1: The development process resembled the waterfall model.
3.2 Disassembling 68k Machine Code

Writing a disassembler served many purposes. It performs instruction decoding, which is one of the vital parts of an emulator. Implementing this familiarised me with the instruction set and addressing modes, and the feel for how the 68k architecture operates. To help with testing and debugging, the instruction pretty-printing routines developed were used in the emulator to show the current instruction; later in development, errors in the decoding of instructions were often first noticed in the disassembler output.

3.2.1 Overview

As described in Section 2.1.1 (page 4), instructions are stored as an instruction word followed by zero or more extension words, plus any additional words needed for immediate data or addressing. The first instruction word, which I shall often refer to as the opcode, provides all the information necessary to know what the instruction is, and how many extension and data words it uses.

The disassembler implemented in this section is structurally very similar to the core emulator. It uses the following algorithm:

1. Read the opcode from the current address.
2. Identify which instruction it encodes.
3. Call the appropriate pretty-printing function.
4. Advance the current address by the length of the instruction, and repeat from 1.

3.2.2 Typedefs

The 68k architecture names three main supported datatypes: bytes, words, and longs (also referred to as double-words). To keep my code clear and easy to read I decided from the outset to define typedefs for each one.

The size of the built-in C integer types is compiler-specific; the basic type int is 32-bit on most systems, but is often 16-bit in embedded platforms, or may be 64-bit in compilers optimised for modern processors. Fortunately the C standard header stdint.h provides typedefs for integers of definite sizes – 8-bit, 16-bit, 32-bit, and beyond – independent of the compiler, in both signed and unsigned forms. Thus I created types.h with the following definitions:

```c
#include <stdint.h>
typedef int8_t sbyte;
typedef int16_t sword;
typedef int32_t slong;
typedef uint8_t ubyte;
typedef uint16_t uword;
typedef uint32_t ulong;
```
### 3.2.3 Identifying Instructions

The disassembler begins by identifying which instruction the opcode represents. This is done by means of an `identify` function, which takes the opcode word as an argument. The first (most-significant) nibble is constant for each instruction, and so to break the function into manageable sections I use it in a switch statement, with 16 cases. The opcode is then compared with various bit-masks using binary AND operations to determine which instruction it encodes, and the function returns a value from an `enum` of types of instruction. The following code snippet demonstrates its structure:

```c
enum opcode_types {
    opcode_ADD,
    opcode_ADDA,
    opcode_ADDI,
    ...,
};

int identify(uword opcode) {
    switch (opcode & 0xf000) {
        case 0x0000:
            if ((opcode & 0xff00) == 0x0600) {
                return opcode_ADDI;
            } else if (...){
                ...
            }
            break;
        ...
    }
}
```

While this is glaringly inefficient – the last `else` block in an `if` structure isn’t reached until all the previous conditions have been checked – this was the quickest and easiest route to getting something that works. I shall consider improving it during the project’s performance evaluation (see Section 4.1, page 35).

### 3.2.4 Pretty-printing

A disassembler’s aim is to translate machine code into assembly code – mnemonic instructions and readable operands. So far we know which instruction an opcode represents, by means of numbering every instruction in an `enum` and returning one of those numbers from `identify`. In order for this to be useful, human-readable text must be generated using the instruction’s mnemonic and its operands.

To compensate slightly for the inefficiency of instruction identification, this was implemented using a table of function pointers to pretty-printing functions, indexed by the instruction type number returned from `identify`. The table was declared as

```c
int (*print_opcode_by_id[]) (uword *, char *) = { ... };
```
The opcode-printing functions return the length of the instruction, in words. This was then be used in step 4 of the disassembly algorithm to advance the current address.

A function `print_ea` was created for pretty-printing effective addresses as displayed in the Syntax column of Table 2.1 (page 3), since this is a very common operation. Many instructions share the same binary format, and therefore I defined a number of macros that deal with the common cases. An example is shown below, which is used for instructions that take an effective address as their operand, such as `JMP` and `PEA`.

```c
#define PRINT_OPCODE_WITH_EA(fn, op, ez) PRINT_PROTO(fn) { 
    char temp[40]; 
    uword ea = (data[0] & 0x3f00) >> 8; 
    int ea_words = print_ea(ea, ez, data, temp); 
    sprintf(buffer, "%-7s %s", op, temp); 
    return 1 + ea_words; 
}
```

This simply extracts the 6-bit effective address field, pretty-prints it to `temp`, and prints out the result after the instruction mnemonic `op`.

3.3 Turning it into an Emulator

In the previous section I described how instructions were decoded and an appropriate pretty-printing function was called. This functionality was used as the basis of the emulator, replacing calls to print functions with calls to execution functions that replicate the operational semantics of each instruction.

3.3.1 Register File and Internal State

The excerpt below shows the register file structure as stored by the emulator. The program counter has a backup copy, PC2, which is set when an instruction is fetched and used when calculating PC-relative addresses. This was necessary because the real program counter will be modified if any extension words are read before a PC-relative address is resolved.

```c
struct {
    ulong PC;     // 32-bit program counter
    ulong PC2;    // program counter backup
    ulong SSP;    // supervisor stack pointer
    ulong USP;    // user stack pointer
    slong D[8];   // 32-bit data registers
    ulong A[8];   // 32-bit address registers
    uword SR;     // 16-bit status register
    uword SR2;    // status register backup
    uword IR;     // instruction register
} regs;
```
The data and address registers were implemented as arrays so the register number fields of instructions could be used directly as array indices. The status register is stored as the combination of system byte and CCR in a single word. Multiple getters and setters were created to help manipulate the status bits. The backup status register is used to store a temporary copy of SR during the exception handling procedure described in Section 2.1.2 (page 6). The last register, IR, was used to store the current instruction word, and was more useful for debugging than anything else.

Not mentioned yet are the two stack pointers, SSP and USP. Side-effects of the setter for the supervisor (S) status bit were employed to seamlessly swap these values into A7. When the S bit is cleared, the current A7 is written to SSP and replaced with USP. When the S bit is set, the converse happens.

The state of the processor was represented using an int taking values from the following enumeration.

```c
enum {
    M68K_STATE_STOPPED,    // the processor is idle
    M68K_STATE_RUNNING,    // the processor is executing instructions
    M68K_STATE_ERROR       // an unrecoverable error occurred
};
```

### 3.3.2 Decoding Effective Addresses

It was realised early on that a robust and general-purpose interface for handling effective addresses would be required, which after several design iterations was implemented as the following methods.

```c
void decodeEffectiveAddress(uword ea_mode_reg, uword op_size);
void readEffectiveAddress(T *ptr);
void writeEffectiveAddress(T value);
```

Here T is used to indicate function overloading, and may be substituted with types ubyte, uword, and ulong, giving seven methods in total.

The `decodeEffectiveAddress` method takes two arguments: the first is the 6-bit effective address field, containing the mode and register number, and the second is the operation size. The constants `OP_SIZE_BYTE`, `OP_SIZE_WORD`, and `OP_SIZE_LONG` were defined for this purpose, having values that mirror those used in the opcode size field. The mode and size are then used to populate a struct with the address type – either data or address register, memory address, or literal value – and the address value. For a memory address, this is the absolute address in memory, while for literals, it is the literal value. For registers it is undefined.

The `read`- and `writeEffectiveAddress` methods simply look at the address type in that struct, and take the appropriate action to move data to or from it. For registers, read or write to the register. For memory addresses, read or write to memory. Literal values are read-only, and trying to write to one prints an error message.
3.3.3 Conditional Operations

There are 16 conditional tests. Their bit patterns and predicates they test are shown in Table 2.2. I implemented these tests as a method, checkConditionCode, which, given the 4-bit condition code, performs the desired test on the status register bits and returns true if the condition is met, or false otherwise. The code excerpt below shows its structure. Constants with the prefix M68K_CC_ were defined for each condition to improve legibility.

```cpp
// Check the CCR according to the 4-bit condition code number
bool M68k::checkConditionCode(int number) {
    switch (number) {
    case M68K_CC_F: // false
        return false;
    case M68K_CC_T: // true
        return true;
    ...
    case M68K_CC_GT: // greater than
        return (!getStatusZ()) && (getStatusN() == getStatusV());
    case M68K_CC_LE: // less than or equal to
        return getStatusZ() || (getStatusN() != getStatusV());
    }
}
```

3.3.4 Fetching and Executing Instructions

Central to any von Neumann processor is the fetch-decode-execute cycle. This was implemented as the method fde, which emulates a single instruction cycle and closely resembles a single iteration of the disassembler loop. The implementation is shown below.

```cpp
void M68k::fde() {
    // fetch and decode the instruction
    regs.IR = fetchWord();
    int id = identify(regs.IR);
    // save current PC for use with PC-relative offsets
    regs.PC2 = regs.PC;
    // execute it
    OpcodeMethodType op = mOpTable[id];
    if (id >= 0 && op != NULL) {
        (this->*op)(regs.IR);
    } else {
        printf("Opcode not implemented! %04x\n", regs.IR);
    }
}
```

A number of support methods were created for common processor operations: fetchWord, that read data from the current PC and increment it; and pushWord and popWord and equivalent long variants, that write and read from the stack respectively using the stack pointer A7.
Memory operations were handled using a MemoryInterface class, that provides read and write methods. This in turn invokes equivalent methods on a MemoryStore object, which represents the memory as an array of bytes and performs endian-swapping when writing words or longs.

Since the core emulator was a standalone C++ class, each instruction was implemented as a method. The preprocessor was used to generate the method names and signatures, since there were tens of methods of identical type. The parameter list and return type were defined in one place, so they could be easily modified for all methods at once.

```cpp
#define M68k_opcode_name(x) op_##x
#define M68k_opcode_parameters uword opcode
#define M68k_opcode_return_type void
#define M68k_opcode_prototype(x) \
    M68k_opcode_return_type M68k_opcode_name(x)(M68k_opcode_parameters)

class M68k {
    ...
    typedef M68k_opcode_return_type (M68k::*OpcodeMethodType) (uword);
    static OpcodeMethodType mOpTable[];
    ...
}
```

The actual lookup table was defined in a separate file so as not to clog up the main header. An excerpt is shown below. The macro M68k_opcode_func_ptr was defined to output a method function pointer given only the name of the opcode, which made the table far more concise.

```cpp
#include "m68k.h"
#define M68k_opcode_func_ptr(x) &M68k::M68k_opcode_name(x)

M68k::OpcodeMethodType M68k::mOpTable[] = {
    // ARITHMETIC
    M68k_opcode_func_ptr(add),
    M68k_opcode_func_ptr(adda),
    M68k_opcode_func_ptr(addi),
    ...
}
```

Obtaining a method pointer requires specifying its class name, but is otherwise not too different to obtaining a function pointer. Calling it differs though: because (non-static) methods act on the . The C++ syntax for this is (this->*methodPtr)(args);

### 3.3.5 Exceptions and Interrupts

All that remains to discuss is exception processing. Two methods were created, with the following prototypes. A number of constants were declared for the exception vector numbers, with names prefixed with M68K_VEC.
void exception(int number);
bool interrupt(int level, int vector);

The `exception` method implements the exception-processing procedure detailed in Section 2.1.2 (page 6). Some additional checks are performed. If the a reset exception is raised, a separate `reset` method is invoked instead, since resetting is logically different. Additionally if certain other exceptions are raised, including bus error or address error, the emulator is put into the stopped state for debugging.

The `interrupt` method combines the 68k's interrupt acknowledge cycle and subsequent exception generation into one function call. The method returns `true` if and only if the exception was acknowledged. First the `level` is checked against the interrupt priority level (IPL) in the status register to check it isn’t masked. A second case is added if the interrupt is level 7, as this indicates a non-maskable interrupt (NMI). The `exception` method is then called, passing the `vector` argument. A special case was added if the vector is zero, to mimic the 68k’s auto-vector feature: the vector number is then generated by adding the interrupt level to `M68K_VEC_INTERRUPT_BASE`. Finally the IPL is set to the new interrupt level, and if the core is in the stopped state it is “woken up” by the interrupt by setting the state to `M68K_STATE_RUNNING`.

### 3.4 DragonBall Peripherals

In the third phase of my project, emulation of the wider DragonBall hardware was added around the core emulator. Figure 3.2 gives an overview of the classes involved. A framework to support the on-chip peripherals was created, and a number of classes the emulate each of these peripherals. A null peripheral stub class was also created to cover those that were not implemented in the project.

The DragonBall has a more complex memory system than the core emulator considered previously; regions of the address space must be dynamically mapped to different devices by the chip-select logic. The construction of the `MemoryInterface` class that performs this will be described.

#### 3.4.1 The Peripheral Framework

Designing the framework for peripherals was one of the more demanding parts of the project. Each peripheral has a bank of registers that must be accessible through memory mapping. At the same time, many of these registers have side-effects when written to.

The solution was implemented using three classes. A new `MemoryInterface` subclass handles address mapping. The register banks of all peripherals are maintained in a `DragonballRegs` class, which extends `MemoryStore`. Lastly, peripherals themselves are represented using a `DragonballPeripheral` base class. Read and writes to the register bank are handled by `DragonballRegs`, which then notifies the peripheral that was affected so that it may execute its side effects.

The `DragonballPeripheral` class forms the superclass of the peripheral implementations, and provides a number of abstract methods that can be seen in its class definition.
Figure 3.2: Class diagram for the DragonBall emulator. The DragonballPeripheral subclasses are not all shown.

class DragonballPeripheral {
protected:
    Dragonball *mHost;
    void *mRegisterPtr;
public:
    DragonballPeripheral(Dragonball *host, ulong base);
    virtual ~DragonballPeripheral() {};
    virtual void reset()=0;
    virtual void clock() {};
    virtual void readByte(ulong *address)=0;
    virtual void readWord(ulong *address)=0;
    virtual void writeByte(ulong address, ubyte value)=0;
    virtual void writeWord(ulong address, uword value)=0;
};

The constructor initialises the two member variables. mRegisterPtr is set to a pointer into the DragonballRegs memory space at the offset specified by the base argument. This provided direct access to the register contents, which was typically done via struct pointers called mRegs or mRegBits.

The reset method is invoked on all peripherals when the DragonBall is reset, and was used to load default register values. clock is invoked for every clock cycle. This generally won’t be used unless the peripheral does something that is timing-related or independant of register accesses, which is why an empty implementation is provided in DragonballPeripheral.

There are then four functions resembling the memory access methods found in the MemoryInterface class. Notice however that the read methods do not return anything – these four functions act as notifications, rather than being used to read and write data, which is handled by DragonballRegs.

The subclass of MemoryInterface, which I called DragonballMemory, was made that
handles mapping of the address space to memory-mapped devices, which were represented as MemoryStore objects. Its definition is shown here:

class DragonballMemory: public MemoryInterface {
    MemoryStore *mMemRegs, *mMemA, *mMemB, *mMemC, *mMemD;
    ... 
public:
    MemoryStore *translateAddress(ulong *address, int *flags);
    ubyte llReadByte(ulong address);
    uword llReadWord(ulong address);
    void llWriteByte(ulong address, ubyte value);
    void llWriteWord(ulong address, uword value);
    ...
};

Address mapping is performed by the translateAddress method, which returns one of the five member MemoryStores or NULL if the address is unmapped.

The four ll-prefixed memory access functions wrap calls to translateAddress, and inspect the result. If it’s NULL, a bus error exception is triggered. Otherwise, the corresponding read or write method on the returned MemoryStore is invoked. The usual non-prefixed memory access functions inherited from MemoryInterface call these lower-level access functions (ll is an abbreviation of this) after validating the input address.

The rest of the translateAddress implementation depends heavily on the chip-select and system control modules, and is therefore discussed in the next section.

### 3.4.2 Chip Select and System Control Registers

The classes PeriChipSelect and PeriControl were created for the chip-select and system control peripherals respectively, both extending the DragonballPeripheral class.

Chip-select ranges have two useful properties: they are a power of two in size, and the base address must be aligned on a multiple of the size. This means that testing whether an address is in range can be done using only binary operators: if the high bits of an address match the high bits of the chip-select’s base address, the address is in the chip-select’s range. This is likely to be faster than comparisons.

Every chip-select had several member variables associated with it that stores precomputed information. CSD will be used as an example. When the peripheral registers are written to these are updated to reflect any changes, taking into account the control bits that have effects of them.

mCSDEnabled = mRegBits->CSD.EN;
if (mCSDEnabled) {
    // size and masks
    int siz = mRegBits->CSD.SIZ;
    if (mRegBits->CSD.DRAM && mRegBits->CSCTRL1.DSIZ3)
        siz |= 8;
    mCSDSize = CS_SIZE_CD[siz]; // look up the size in table
    mCSDLowMask = mCSDSize-1; // all bits below the size
}
mCSDHighMask = ~mCSDLowMask; // all other bits

// base address
mCSDBaseAddr = (ulong) mRegs->CSGBD << 13;
if (mRegBits->CSUGBA.UGEN)
    mCSABaseAddr |= (ulong) mRegBits->CSUGBA.AGBA << 29;

// validate base address
if (((mCSDBaseAddr & mCSDHighMask) != mCSDBaseAddr) {
    printf("Warning: CSD base address not aligned\n");
}

// memory protection
siz = mRegBits->CSD.UPSIZ;
if (mRegBits->CSCTRL1.EUPEN && mRegBits->CSCTRL1.DUPS2)
    siz |= 4;

mCSDUnprotected = mCSDSize >> siz;

PeriControl was relatively simple, as it only needed to provide the contents of the system control registers to other components. This was done by declaring friend classes, which gives those classes access to private members, and thus the mRegBits pointer.

We now have the necessary classes to look at the memory mapping in the DragonballMemory class. The excerpt below shows how the given address is compared against the register file and CSD address ranges.

MemoryStore *translateAddress(ulong *address, int *flags) {
    // start off with no flags set
    *flags = 0;
    // see if address maps to the Dragonball register region
    ulong mRegHighMask = mSysControlPtr->mRegBits->SCR.DMAP ?
        DRAGONBALL_REGISTERS_BASE : DRAGONBALL_REGISTERS_BASE2;
    if ((*address & mRegHighMask) == mRegHighMask) {
        *address &= DRAGONBALL_REGISTERS_MASK;
        return mMemRegs;
    }
    // see if it maps to Chip Selects A to C
    ...
    // see if it maps to Chip Select D
    if (mChipSelectPtr->mCSDEnabled) {
        if (((*address & mChipSelectPtr->mCSDHighMask) ==
            mChipSelectPtr->mCSDBaseAddr) {
            // the address matches the range. mask it to make it local
            *address &= mChipSelectPtr->mCSDLowMask;
            // check memory protection
            if (mChipSelectPtr->mRegBits->CSD.RO)
                *flags |= READ_ONLY;
            if (*address >= mChipSelectPtr->mCSDUnprotected) {
                if (mChipSelectPtr->mRegBits->CSD.ROP)
                    *flags |= PROT_READ_ONLY;
                if (mChipSelectPtr->mRegBits->CSD.SOP)
The flags are used to pass information about memory type and protection to the caller. If the processor is not in supervisor mode when the PROT_SUPERVISOR_ONLY flag is set, or an attempt is made to write when one of the READ_ONLY is set, a bus error exception is raised.

3.4.3 The Interrupt Controller

The interrupt controller keeps track of and prioritises interrupts generated both within the DragonBall and from external sources. The PeriInterrupt class was created with the following definition:

```cpp
class PeriInterrupt: public DragonballPeripheral {
    friend class DragonballMemory;
    int mHighestLevel;
    ...

    public:
    ...
    void interrupt(int number);
    void clearInterrupt(int number);
    void postInterrupts();
};
```

Every interrupt was defined as a constant, whose value was chosen to correspond to its bit position in the interrupt registers. The INT_BIT macro converts the number to the bit, allowing such expressions as IPR |= INT_BIT(INT_TMR1) to be written. A look-up table was also defined that converts interrupt number to priority level; the list of values is given on page 9-1 in the MC68VZ328 User’s Manual [3]. The table was not declared const because some interrupts have configurable levels.

```cpp
#define INT_BIT(x) (1 << (x))
enum {
    INT_SPI2 = 0,
    INT_TMR1,
    INT_UART1,
    INT_WDT,
    ...
};
int INT_LEVEL_TABLE[32] = { 4, 6, 4, 4, 4, 3, 4, 6, ... };
```
Interrupts signals are manipulated via the `interrupt` and `clearInterrupt` methods, which assert and deassert an interrupt respectively. Each also updates the member variable `mHighestLevel`, which is used to keep track of the highest level interrupt present in the IPR.

Lastly a `postInterrupt` method was created, which is called at the end of the DragonBall’s instruction cycle. It inspects the value of `mHighestLevel`, and if it is non-zero invokes the `interrupt` method on the DragonBall’s M68k instance.

### 3.4.4 Serial Peripheral Interfaces

There are two serial peripheral interfaces, named SPI1 and SPI2. Only SPI2 was required by the Palm and will be covered in this section.

A brief introduction to SPI was given in Section 2.2.4 (page 8). The general principle is that bits are exchanged between a master and slave device one at a time. An abstract class for SPI devices was made with a single method to facilitate this functionality. For SPI2, the DragonBall is configured as master, and the external device as slave.

```cpp
class SpiInterface {
public:
    virtual uword exchangeBit(uword bit) = 0;
};
```

The `exchangeBit` method is called by the master for every DCLK cycle, passing its DOUT in the argument and reading its DIN from the return value. On the receiving end, the slave reads its DIN from the argument, and puts its DOUT in the return value.

SPI2 was implemented in the `PeriSPI2` class, with the following definition:

```cpp
class PeriSPI2: public DragonballPeripheral {
    SpiInterface *mSpiDevice;
    int mSpiClockDivider;
    int mClockCounter;
    int mBitsRemaining;

    void updateSpiCont2(uword value);
    void startExchange();
    void exchangeBit();
    void exchangeComplete();
    void sendSpiInterrupt();

public:
    ... 
    void connect(SpiInterface *device);
};
```

The `connect` method is used to set the `mSpiDevice` member. When an exchange is triggered by a write to the XCH status bit, the `startExchange` method is called, checks the number of bits that will be transferred and aligns the data register ready
to be exchanged. The `exchangeBit` method is the master device counterpart of the `SpiInterface`’s slave `exchangeBit`, and moves a bit out of one end of the peripheral’s data register and brings the received bit into the other.

```cpp
void PeriSPI2::exchangeBit() {
    // exchange one bit between SPI2 and the peripheral
    uword tx = mRegs->SPIDATA2 & 0x8000;
    uword rx = mSpiDevice->exchangeBit(tx);
    // shift the MSB out
    mRegs->SPIDATA2 <<= 1;
    // and the new bit in
    if (rx) mRegs->SPIDATA2 |= 1;
}
```

Once the requested number of bits have been exchanged, `exchangeComplete` is invoked, which realigns the received data in the data register, and if the interrupt enabled bit is set calls the `sendSpiInterrupt` method, which asserts the SPI2 interrupt.

### 3.4.5 General Purpose Timers

The DragonBall has two general purpose timers, which can be configured to count a certain number of clock cycles and send and interrupt. These were implemented in the single `PeriTimers` class.

The main method of interest is the `updateTimer` method, which takes the timer’s index – 0 or 1 for TMR1 or TMR2 respectively – and increments its counter according to the selected prescaler. This is invoked every clock cycle for both timers.

```cpp
void PeriTimers::clock() {
    updateTimer(0);
    updateTimer(1);
}

void PeriTimers::updateTimer(int index) {
    PeriTimerRegBits *regs = &mRegBits->T[index];
    // check the timer is enabled
    if (!regs->TCTL.TEN)
        return;
    if (regs->TCTL.CLKSOURCE == 0)
        return;
    // prescale the clock signal
    if (mPrescalerCounter[index] < regs->TPRER.PRESCALER) {
        mPrescalerCounter[index]++;
        return;
    }
    mPrescalerCounter[index] = 0;
    // increment the timer’s counter
    regs->TCN.COUNT++;
    // compare with COMPARE register
```
if (regs->TCN.COUNT == regs->TCMP.COMPARE) {
    // reset the counter if in restart mode
    if (regs->TCTL.FRR == TIMER_MODE_RESTART)
        regs->TCN.COUNT = 0;
    // set the compare bit in the status register
    regs->TSTAT.COMP = 1;
    // generate an interrupt if enabled
    if (regs->TCTL.IRQEN) {
        int i = index == 0 ? INT_TMR1 : INT_TMR2;
        mHost->getInterruptController()->interrupt(i);
    }
}

3.4.6 Real-time Clock

The real-time clock module provides a number of functions, including interrupts every
at regular intervals ranging from fractions of seconds to a day, and keeping track of the
time of day and the date. It was implemented in the PeriRTC class, shown below.

class PeriRTC: public DragonballPeripheral {
public:
    ...
    void tick();
    void clock();
    void setTime(int hours, int minutes, int seconds);
    ...
};

Two levels of emulation were used here, unlike other peripherals. The usual clock
method was used to emulate the clock cycle-dependant real-time interrupts. A second
tick method was added that emulates the second counter, so that the emulator’s sense
of time could be made independant of its running speed by ticking on a separate thread
that accurately calls tick at one second intervals.

The implementation of tick, shown below, is essentially a ripple counter over each
register. When one of the counters reaches its maximum value, it may trigger an interrupt
if that interrupt is enabled.

void PeriRTC::tick() {
    uword interrupts = 0;
    if (!mRegBits->RTCCTL.RTCEN) // check the RTC is enabled
        return;

    if (++mRegBits->RTCTIME.SECONDS >= 60) { // increment seconds
        mRegBits->RTCTIME.SECONDS = 0; // minute roll-over
        if (++mRegBits->RTCTIME.MINUTES >= 60) {
            mRegBits->RTCTIME.MINUTES = 0; // hour roll-over
            if (++mRegBits->RTCTIME.HOURS >= 24) {
            }
        }
    }
}
mRegBits->RTCTIME.HOURS = 0; // day roll-over
++mRegBits->DAYR.DAYS;
interrupts |= RTC_INT_DAY; // fire day interrupt
}
interrupts |= RTC_INT_HR; // fire hour interrupt
}
interrupts |= RTC_INT_MIN; // fire minute interrupt
}
interrupts |= RTC_INT_1HZ; // fire second interrupt

if (mRegs->RTCTIME == mRegs->RTCALRM) // check alarm
interrupts |= RTC_INT_ALM; // fire alarm interrupt

interrupts &= mRegs->RTCIENR;
if (interrupts) { // dispatch any interrupts
mRegs->RTCISR |= interrupts;
mHost->getInterruptController() -> interrupt(INT_RTC);
}
}

Interrupts were accumulated in a bitfield, which was ANDed at the end to test all generated interrupts with the interrupt-enable bits at once, rather than having to check at each interrupt generation point. This made the code more efficient and clearer to read.
3.5 Palm Hardware

While the DragonBall provides many on-chip functions, a Palm device contains additional off-chip hardware, including memory, a dedicated display controller, USB interface, and touch screen controller. I created software implementations of this hardware that could be used with the DragonBall emulator to form a Palm emulator. This was the first extension completed, and enabled the emulator to boot Palm OS.

3.5.1 RAM and ROM

The m515 has a 4 MB ROM chip, which stores the operating system and bootloader, and a 16 MB DRAM chip, used as the main storage memory. These were implemented as two subclasses of MemoryStore, named ROMStore and RAMStore respectively.

The RAMStore class functionality was already implemented by MemoryStore, so no further modifications were necessary. The ROMStore in contrast represents read-only memory, and thus overrides the writeByte and writeWord methods, replacing them both with calls to a private writeAttempt method. This currently displays a warning message, since writes to ROM are probably erroneous and indicate a bug.

3.5.2 The Display Controller

The m515’s display controller appears as a memory-mapped device, and therefore was implemented as a subclass of MemoryStore.

```cpp
class LcdController: public MemoryStore {
    ubyte *mRegStore;
    ubyte *mLUT;
    ...
    public:
    ...
        const ubyte *getScreenBufferPtr();
        const ubyte *getLUTPtr();
        uword getDisplayWidth();
        uword getDisplayHeight();
        ulong getDisplayLineStride();
        int getDisplayBitsPerPixel();
        void setVerticalDisplayPeriodBit(bool displayPeriod);
        void exportToBitmap(const char *filename);
};
```

Its constructor allocates an 80 KB MemoryStore for the display memory. Additional memory is allocated to mRegStore for the registers and to mLUT for the colour look-up table. During memory accesses, address bit 17 to dispatch the read and write commands to separate handlers for the display memory, which involved simply invoking the superclass’s implementation, and the register file. Reads and writes to the lookup table were implemented in the writeByte to achieve the behaviour described in .
The various getters provide convenient access to important display parameters, performing any calculations necessary as described in Section 2.3.2. A setter for vertical display period bit also added so that it could be toggled during screen redrawing, as would happen in the real hardware. This helped to reduce flicker.

Lastly a method was added that allowed the screen image to be dumped to a Windows Bitmap file. This was used to produce Figure 3.3 on page 30.

### 3.5.3 The USB Controller

The USB controller also interfaces as a memory-mapped device and was implemented as a subclass of `MemoryStore`. Something that sets it apart is that it has an 8-bit data interface. Support for this was not implemented during the course of the project, but its interface was sufficiently simple that it could be adjusted to suit normal 16-bit operation, since it comprises only two 1-byte registers.

```cpp
class UsbController: public MemoryStore {
    uword lastCommand; // transaction info
    uword state;

    UsbControllerRegs mRegs; // internal state
    UsbControllerRegBits *mRegBits;
    UsbEndpoint mEndpoint[6];
    UsbEndpoint *mSelectedEndpoint;
    uword mSelectedEndpointIndex;

    void startCommand(ubyte cmd); // private methods
    void processWrite(ubyte data);
    ubyte processRead();

    ...
};
```

What it lacks in its interface is made up by the increased state necessary for keeping track of transactions and maintaining USB queues. A complete implementation of every transaction was not made during the project, but the basic framework for processing a transaction was, and all commands were recognised.

Transactions are initiated by writing to the command register, which invokes `startCommand`. This sets up `lastCommand` and `state` for the transaction. Subsequent reads and writes to the data register get passed to the `processRead` and `processWrite` methods.

### 3.5.4 The Touchscreen Controller

A new class was created called `TouchScreenAdc` with the definition shown below.

```cpp
class TouchScreenAdc: public SpiInterface {
    // internal state for emulation
    int mInState; // number of bits we've received
```
int mOutState; // number of bits left in response
ubyte mCommand; // received command
uword mResponse; // response to send back
// power settings
bool mEnableADC; // enable ADC
bool mEnableRef; // enable internal reference voltage
// the various inputs to the ADC
uword mInXplus, mInXminus; // digitiser X+, X-
uword mInYplus, mInYminus; // digitiser Y+, Y-
uword mInVBAT, mInAUX; // battery voltage and auxiliary input
uword mInTEMPO, mInTEMP; // temperature
// internal functions
void processCommand();

public:

uword exchangeBit(uword bit);
void setTouchCoords(uword x, uword y);
...

};

Each input channel to the ADC is represented by a member variable. The exchangeBit
method inherited from SpiInterface was implemented as a simple state machine, shown
below. Because the ADC must be capable of receiving a command at the same time as
returning data, separate variables were used to keep track of the current state for both
input and output.

uword TouchScreenAdc::exchangeBit(uword bit) {
  uword tx = 0;
  // is there any data to send back?
  if (mOutState) {
    tx = mResponse & 0x8000; // shift bit out
    mResponse <<= 1;
    mOutState--;
    // accept the next command when there are 6 bits left
    if (mOutState == 6)
      mInState = 0;
  }
  // process incoming bit
  if (mInState == 0) {
    if (bit) { // command starts on first high bit
      mInState = 1;
      mCommand = 1;
    }
  } else if (mInState < 8) {
    mInState++;
    // shift it into the command
    mCommand <<= 1;
    if (bit) mCommand |= 1;
    if (mInState == 8) // process it if we have all the bits
      processCommand();
  } else if (mInState == 8)

} else {
    // the ADC is busy and we do not accept new commands
}
return tx;
}

The private `processCommand` method selects the response to the command from the `mIn*` members, and updates the power setting from the command.

### 3.5.5 Putting It All Together

Using the components implemented up to this point, describing the Palm m515 in software was very easy:

```c
// set up the Palm hardware
Dragonball dragonball;
ROMStore rom("m515.rom");
RAMStore ram(16*1024*1024);
LcdController lcd;
UsbController usb;
TouchScreenAdc adc;

dragonball.setStoreCSA(&rom);
dragonball.setStoreCSB(&lcd);
dragonball.setStoreCSC(&usb);
dragonball.setStoreCSD(&ram);
dragonball.getSPI2()->connect(&adc);
dragonball.reset();
```

Emulation is then started by invoking the `run` or `step` methods of the `Dragonball` instance. Figure 3.3 shows the first signs of success.

![Palm OS boot screen capture](image)

**Figure 3.3:** Capture of the Palm OS boot screen, created using the display controller’s `exportToBitmap` method.


### 3.6 Android Details

Up until this point, the emulator’s host platform has been ignored, and the emulator has had no user interface. This section will briefly describe the Android app that was created for the emulator.

The Android app was written in Java, and interacts with the emulator through the Java Native Interface (JNI). Methods of a Java object are declared with the `native` keyword, and a C++ function with a corresponding name will be called when the message is invoked. All the emulator functionality was wrapped in the Java `Emulator` class, whose source is shown here.

```java
public class Emulator {
    static {System.loadLibrary("dragon");}
    // states corresponding to those in Dragonball.hpp
    public static final int STATE_RUNNING = 0;
    public static final int STATE_PAUSED = 1;
    public static final int STATE_ERROR = 2;
    // pointer to native objects
    private long nptrResources;
    // emulator instance lifecycle methods
    public static native Emulator create(Context context, String rom);
    public native void destroy();
    // control
    public native void run();
    public native void stop();
    public native void reset();
    public native int getState();
    public native String getErrorMessage();
    public native boolean isRunning();
    public native boolean isVRamDirty();
    // interaction
    public native void getScreenImage(Bitmap bitmap);
    public native void setDigitizerCoords(float x, float y);
    public native void setDigitizerPressed(boolean pressed);
}
```

Java does not allow native constructors, so the `create` method is used as a substitute. This method allocates all the emulator objects as per Section 3.5.5. The pointer to the resulting C++ object is cast to a `long` and stored in `nptrResources` so that it can be accessed again in other methods – Java classes cannot have native members, so this is a necessary evil. Java `longs` are 64-bit, so there is no danger of pointer truncation.

The app provides two `Activity`s, one to manage emulator sessions, the other to run the actual emulator. Screenshots of these are shown in Figure 3.4. The main emulator user interface consists of a `View` for the display, another `View` for the silkscreen area, and two Action Bar buttons that allow the user to pause or reset the emulation. Both `Views` process touch events and invoke the `setDigitizerCoords` and `setDigitizerPressed` methods of the `Emulator`, which act as described in the next section, Pen Input.
The emulator is run in three threads. The first is a `EmulatorThread` runs the main emulation loop until it is paused. The second is a `DisplayThread` which, every 20 ms, invokes the `getScreenImage` method and refreshes the display `View`. The third handles the real-time clock, invoking its `tick` method every second.

![Figure 3.4: Screenshots of the Android app created for the project. The left shows the session list, and the right shows the emulator running.](image)

### 3.6.1 Pen Input

Adding support for pen input through the touchscreen was one of the most satisfying stages of the project – the emulator was no longer just a thing to watch processing instructions, but could be interacted with.

When the stylus is pressed on the touchscreen digitiser, IRQ5 is asserted, and when it is removed, IRQ5 is deasserted. The coordinates of the touch are read from the analogue-digital converter when the XPOS and YPOS lines are selected. Thus pen input is as simple as invoking the `interrupt` and `clearInterrupt` methods on the interrupt controller when Android touch events are received, and manipulating the ADC’s readouts via the `setTouchCoords` method.
This information had to be reverse engineered. IRQ5 was deduced from the fact that it is unmasked – that is, its bit is cleared in the IMR) in the PenInit function call during boot. Conversion of raw ADC values to screen coordinate was done by disassembling the PenRawToScreen function. The inverse of the function could then be applied to the touch coordinates in the emulator to set the ADC readouts.

3.6.2 Thread Safety

While using multiple threads is efficient and mirrors the parallel nature of hardware, problems are often introduced the threads need to share a resource. Of the four threads, only the display update thread was initially safe because it performs only read operations.

The other three – the main emulator, UI (for pen input), and RTC threads – have a common resource that is the interrupt controller. It was necessary to prevent external interrupts, generated by the pen, from conflicting with writes to the interrupt controller registers and internal interrupts generated on the main thread. The traditional solution is to use a mutex object.

The Android NDK provides the pthreads library, which defines types and functions for mutex object creation and locking. The Dragonball was modified to create a mutex in its constructor (and free it in its destructor), which gets locked for the duration of the instruction cycle. Other threads can then lock the same mutex to ensure they do not interact at the same time.

For better OOP design, two new methods were added to the PeriInterrupt class: externalInterrupt and externalClearInterrupt. These hide the detail of having to lock the mutex while processing the interrupt.

3.7 Summary

I have described how the emulator was implemented using the information collected during preparation. Figure 3.5 shows photograph of the Palm m515 used in this project sitting next to its emulated version on running my phone. In the next chapter I shall evaluate the emulator’s performance and examine how correct the implementation is.
Figure 3.5: Old and new, side-by-side: the Palm m515 used in this project next to its emulated version on an Android smartphone.
Chapter 4

Evaluation

In this chapter I carry out quantitative and qualitative evaluations of my project, which were used to consider how well the project met the corresponding success criteria: performance and correctness, respectively.

4.1 Emulator Performance

The performance of an emulator is critical. While it is perhaps more important when creating emulators for games consoles, a Palm handheld is still a highly interactive system, and running too slowly degrades the user experience. One of my project’s success criteria was that it should not be slower than the physical device, and I aim to quantify its performance by running a number of benchmark programs. The same benchmarks will be run on the physical m515, and comparisons will be made.

4.1.1 Methodology

Performance of the emulator was measured both on my laptop and smartphone in five numbered tests, covering combinations of software versions and hardware:

1. Test the physical Palm m515
2. Test the core-only emulator, running on Windows
3. Test the core-only emulator, running on Android
4. Test the Palm emulator, running on Windows
5. Test the Palm emulator, running on Android

Five benchmark programs were run in each test, which are summarised below:

- **Fibonacci** — Compute the 32nd Fibonacci number using a naïve recursive function
- **Mandelbrot** — Plot the Mandelbrot set in a 160x160-pixel window
- **Loop benchmarks** — Three programs that execute certain types of instruction millions of times: no-op, register arithmetic, and memory arithmetic
More details of each benchmark program, including source code and an explanation of how they were prepared for each test platform, may be found in Appendix A.

The tests were run three times each. The time taken for the test to complete was measured by taking two timestamps, one when the emulation was started and the other on encountering a STOP instruction, and subtracting to get the difference. The average of the three times was recorded. The raw results are presented in Appendix B.

Standard deviations of the results of all tests were calculated, and averaged around 1% with a worst case of 2.3%. This is consistent enough that no error bars were considered.

### 4.1.2 Timing

On Windows, timestamps were obtained using the Performance Counter API, via calls to `QueryPerformanceCounter`. While this provides extremely high-resolution timing, I chose it more for familiarity and its ease-of-use than for its accuracy. On Android, timestamps were obtained using the well-known `System.currentTimeMillis()` function. Calling a Java function from native code does incur some overhead, but because two timestamps are subtracted this should cancel out.

The OS-dependent functions were abstracted away by creating a `Timer` class, using one implementation for Windows builds and another for Android. It exposes a number of simple methods as shown in its definition:

```cpp
class Timer {
public:
    void start(); // records the start time
    double stop(); // records the end time, returns difference
    double getElapsedTime(); // also returns difference
};
```

The following code snippet demonstrates its usage. Preprocessor commands were used to easily toggle the emulator in and out of benchmarking mode. A new method, `runBenchmark`, was added to the Dragonball class, which functions identically to the usual `run` method except that it stops when the M68k stops.

```c
#ifndef BENCHMARKING
    dragonball->run();
#else
    Timer timer;
    timer.start();
    dragonball->runBenchmark();
    timer.stop();
    printf("Benchmark finished: %f s", timer.getElapsedTime());
#endif
```
4.1.3 Results

The average runtimes of the benchmarks are shown in Figure 4.1.

Unsurprisingly, the benchmarks had the shortest execution time when run on my laptop. Benchmarks took the most time to complete in the Palm emulator running on Android, followed by the physical m515.

The core-only emulator always completed faster than the Palm emulator, by a factor of 4.8 on Android and 2.8 on Windows. This was because of the additional overhead in the Palm emulator of memory mapping and emulating the peripherals.

It is very interesting to compare the results from the physical Palm and Palm emulator in the Mandelbrot and no-op benchmarks. The no-op program executes more than twice as many instructions, and yet execution time on the physical Palm was lower in this benchmark. For the Palm emulator it was lower in the Mandelbrot benchmark. There are two factors that cause this. The first is that the Mandelbrot program executes many arithmetic instructions, including multiplications, which are very slow compared to no-ops. The second is the per-instruction overhead. This evidently outweighs the gains from simpler instructions in the emulator, but is low on the physical device, as one might expect – hardware is efficient.

Some further analysis was done by subtracting the time taken in the no-op loop benchmark from the other two loop benchmarks. This leaves the time taken to perform only a read, addition, and write to either register or memory. These times and the ratio between them are shown in Table 4.1.

Figure 4.1: Average execution time of each benchmark program over three runs, in five test environments.
<table>
<thead>
<tr>
<th></th>
<th>Palm</th>
<th>Windows</th>
<th>Android</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register time, ms</td>
<td>1033</td>
<td>184</td>
<td>1256</td>
</tr>
<tr>
<td>Memory time, ms</td>
<td>9573</td>
<td>1164</td>
<td>4480</td>
</tr>
<tr>
<td>Ratio</td>
<td>9.27</td>
<td>6.33</td>
<td>3.57</td>
</tr>
</tbody>
</table>

Table 4.1: Relative time taken to execute a register and memory operation in the on each platform. The ratio shows how many times slower a memory operation was than a register operation. The Palm emulator results were used for the Windows and Android columns.

Both emulators had a lower ratio than the physical Palm, and the lowest ratio was actually when it was run on Android. This is not what I expected at all – I had assumed that in the emulators the numerous tests performed on addresses would make memory accesses slower than the physical device, where the parallelism of hardware can be used to check all chip-selects at once. Perhaps the reason is that the memory chip in the Palm are much slower, and have minimal or no caching, while the modern processors in my smartphone and laptop have extensive caches.

Unfortunately the performance success criterion was not met – the emulator on Android is slower than the physical m515. It is, however, only 1.35 times slower, using the Fibonacci test as the reference, so it’s very close. My smartphone was released in 2014, and now in 2016 another iteration of Moore’s law will have passed, so it is likely that a recent high-end smartphone could run the emulator at full speed.

I shall now consider ways in which the emulator’s performance might be increased. The Android Profiler does not support native code, so I was unable to identify bottlenecks this way. One can however speculate. The core-only emulator ran 2.83 times slower on Android than on Windows, while the Palm emulator ran 5.08 times slower on Android. This difference must be attributed to discrepancies between the two version of the Palm emulator: the Android version has pen input, while the Windows version does not. The only difference in emulation loop is that a mutex is locked and unlocked every instruction cycle, and thus this must, at least partly, be responsible for the slow-down.

Therefore I considered whether it would be possible to remove the reliance on a mutex in the emulation loop entirely, and devised the following idea. A separate variable, call it IXR, could be used exclusively for external interrupts, which would be updated by the externalInterrupt and externalClearInterrupt methods. Only these methods would need to lock a mutex. At the end of an instruction cycle, the bits in IXR would be merged into the IPR, taking care to read IXR only once.

Another optimisation one could make is to make instruction decoding more efficient. Currently this is done in the identify function using many if statements check the opcode against bit patterns, until one is found that matches. This could be improved by using a lookup table with 65536 entries, indexed by opcode, that stores the result of the identify function. Memory versus speed is a common trade-off in software.
4.2 Implementation Correctness

The second half of the evaluation focuses on the qualitative success of the project, with respect to the second success criterion: correctness. Needless to say, if an emulator does not function the same as the hardware it mimics, there will be compatibility issues running real machine code in it. The aim of this section is to ascertain correctness by looking at just that: running software in my emulator and verifying that it behaves correctly.

Dijkstra (1969) made a famous quote: “Testing shows the presence, not the absence, of bugs.” One of the reasons for this is that testing a complex system cannot possibly cover every use-case, and thus bugs can slip through the net. The chances of this can be reduced by using a broader range of tests. I also consider some options for more thorough verification techniques, which would be preferred but were outside the scope of this project owing to time constraints.

4.2.1 Methodology

Palm OS was booted in the Palm emulator. Systematically, every common action the user might perform was tried. Each built-in application was opened, and its buttons and menu items selected. The effect of the action was observed, and whether the emulator behaved as expected was recorded, along with a screenshot. The proportion of actions that failed to produce the expected result was then calculated.

The complete list of actions performed and accompanying screenshots can be found in Appendix C.

4.2.2 Results

112 out of the 116 (96.6%) actions performed produced the expected result. Four produced incorrect results. From this I can be confident that my implementation is largely correct, but clearly there exists at least one incorrect component.

Within my Palm emulator there are three possible causes of failure: the core emulator is incorrect, emulation of a certain piece of hardware is inaccurate, or the emulation is correct but the virtual hardware is not configured in exactly the same way as inside the physical Palm.

Attempting to launch the Date Book application or view the HotSync log resulted in fatal errors within the emulator. The actions were repeated a couple of times to make sure. Sometimes the Palm OS “Fatal Error” popup appeared, and order could be restored by resetting the emulator. However other times the emulator app would crash as a result of \texttt{segv} – segmentation fault. This problem must lie in the core emulator, as inaccurate hardware emulation would not cause such crashes. The tools for debugging NDK applications proved hard to set up, so at the time of writing I have not been able to locate the cause of these crashes.

I was able to trace the emulated control flow in the Windows build of the emulator, and the problem was being caused by a jump instruction whose target address was given as a literal, and that pointed to non-executable data in RAM – the middle of some text,
with no obvious significance. It is possible that this is the result of an instruction decode error, although jumps are so common it would seem unlikely that the problem did not manifest itself sooner.

The third problem was observed on the Info and Delete menu options of the applications home screen: although the m515 has 16 MB storage, only 8 MB was recognised by the OS. The problem is most likely caused by some missing hardware rather than an emulation error. In fact, this was a known issue with the original POSE. See the Known Issues section of the PHEM website [7].

Finally, trying to beam an application locked up the emulation. Probably because it’s waiting for a register bit to flip, and because the hardware is incompletely emulated, it never happens. A similar problem was encountered earlier in development with the USB controller, and was overcome quite by chance by changing a status bit in one of the Port registers. It should be possible to determine what the loop condition is through disassembly of instructions around the program counter when this happens, but this has not been done.

4.2.3 Improved Methods

The method used in the evaluation involved testing only the built-in software of the Palm, as there was no way to install new software. It would be good to test the emulator with more software from third parties – the more software tested, the more certain we can be that the implementation is correct, assuming no bugs are uncovered.

Verification by testing is not guaranteed to find problems, although in this case multiple problems were found. There are two other methods one could employ to get more thorough results: formal verification, and a correctness test benchmark program.

Formal verification is the decisive tool in determining whether a system is correct. Its major drawback is its difficulty. While we can use computers to prove that a program meets a set of verification conditions, those conditions have to be created from assertions placed throughout the program. The program must be understood for this to be done, and it is therefore a task best suited to humans. However, humans are unreliable and tend to make mistakes, so for a complex system one must also show that the assertions are correct.

The second option is a benchmark program that runs in the emulator and rigorously tests every instruction by putting the processor into a certain state before an instruction, and verifying the state is correctly modified after it has executed. It suffers the same drawback as formal verification – the pre- and postconditions have to be created manually by a human who understands the architecture.

4.2.4 Summary

The evaluation reveals that neither of the success criteria were fully met; the Palm emulator runs 1.35 times slower than the physical Palm, and the implementation is not correct, with 3.4% of actions failing. However both results are not bad. Potential ways of improving performance were suggested.
Chapter 5

Conclusions

So far I have described the preparation, implementation, and evaluation of an emulator for the 68k architecture, which has been extended to support Palm OS and can be run within an Android app. In this final chapter I reflect on what I have achieved, what could have been done better in hindsight, and scope for further extension towards a complete Palm emulator.

5.1 Achievements

The original aim of this project was to write an emulator for the Palm m515 that could run on Android, divided into the sub-goals of creating a core 68k emulator, adding peripherals found in the DragonBall system-on-chip, and extensions for Palm. I succeeded in implementing the first two goals, and was able to make a start on the third. Although the project in its current state is by no means a complete emulation of the Palm hardware, I’m very pleased with the results.

The success criteria concerned the emulator’s performance and correctness, and the extend to which these were met was covered during the evaluation. While neither criteria were met in their entirety – there exists at least one bug in the core emulator, and emulation speed on my smartphone is slightly slower than the original – I am nonetheless satisfied. It is incredible to think that in 12 years, mobile hardware has advanced to the point that the old can be decently emulated on the new without paying significant attention to efficiency.

5.2 Review of Project Execution

On the whole I believe the project was completed satisfactorily and my workplan was well-designed, even if I found it hard to keep up at times. Incrementally building on the previous implementation phases proved an effective strategy that allowed me to be familiar with the lower levels, yet abstract them away when working on the next phase.

Perhaps one area that could be improved is applying more OOP principles to the higher level classes; most of the emulation control ended up in the Dragonball class, whereas it would have been more sensible to create a new Emulator class that contains a
Dragonball, along with other hardware classes. In the Android implementation presented this was in part achieved in the JNI layer and the Java Emulator class, but the result was far from modular. More careful planning at the beginning of each implementation phase would perhaps have prevented this.

Only after evaluating did I become aware of the C standard function clock, which could be used in place of the OS-dependent timing methods giving a single standard Timer class. This would have saved some implementation time.

### 5.3 Extensions and Future Work

There is plenty of scope for further work on this project. While the core 68k emulator is largely complete, there is at least one outstanding bug that causes a crash when the Date Book application is launched or the Hotsync log is viewed. Performance also needs to be improved so that the success criterion is comfortably met. Solving these issues would be desirable, and the ideas presented in the Evaluation could form a starting point.

Apart from the real-time clock module, which has a dedicated thread accurately counting seconds, the timing in terms of clock cycles within the emulator is not accurate. Currently each instruction is assumed to take a single clock cycle, but in reality it is usually many more. This is significant because it affects the relative speed of the peripherals to the core; in the current version, timers will decrease too slowly relative to the program execution.

Considering the practical use of the emulator, it lacks the ability to install third-party applications on the emulated Palm. Being able to do this would allow more thorough software compatibility testing. Hewgill describes how it was achieved in Copilot in *Copilot: Design and Development* [11], which could be used as a starting point.

It is also important to remember that a smartphone is battery-powered, and user’s care about battery life. Currently the emulator pays no regard to power usage, and the emulation loop keeps the phone’s processor awake constantly.

Finally, a number of extensions goals were put forward in the project proposal that were not implemented. I shall briefly consider them again:

- **Palm hardware buttons.** There is however not much space on the phone’s screen for more buttons. The only button that would be useful is the power button, which is needed to wake the Palm when it enters doze mode. This would not be necessary if a way to prevent doze mode was found.

- **SD card emulation.** Adding this would also provide the means of installing third-party applications mentioned above.

- **Sound, vibration, and LED notifications.** These are purely aesthetic.
Bibliography


http://perpendox.com/phem/

http://hewgill.com/pilot/copilot/index-old.html

http://onboardc.sourceforge.net/

http://www.easy68k.com/

Appendix A

Benchmark Programs

This appendix provides details of the five benchmark programs used during performance evaluation. Table A.1 provides a summary of each program, including the number of instructions executed in the benchmark.

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci</td>
<td>Computes the 32nd Fibonacci number using a naïve recursive function. Written in C using OnboardC.</td>
<td>95163589</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>Draws the Mandelbrot set in a 160x160 pixel window. Written in C using OnboardC.</td>
<td>11655531</td>
</tr>
<tr>
<td>No-op test</td>
<td>Executes a NOP instruction several million times in a loop. Written in assembly using EASy68k.</td>
<td>25165956</td>
</tr>
<tr>
<td>Arithmetic test</td>
<td>Performs register-bound arithmetic instructions several million times in a loop. Written in assembly using EASy68k.</td>
<td>25165958</td>
</tr>
<tr>
<td>Memory test</td>
<td>Performs memory-bound instructions several million times in a loop. Written in assembly using EASy68k.</td>
<td>25165959</td>
</tr>
</tbody>
</table>

Table A.1: Summary of the benchmark programs.

A.1 Construction Details

The first two programs, Fibonacci and Mandelbrot, were written in C and compiled on my physical m515 using the Onboard Suite [9]. Their source code is shown in Listings A.1 and A.2 respectively. OnBoardC’s #pragma segment directive was used to compile them as stand-alone code resources, which were then extracted using a hex editor.

The three single-instruction tests (no-op, arithmetic, memory) were written in 68k assembly and assembled using EASy68k [10]. The assembler outputs S-record files, which are typically then “burned” to ROM chips – they contain a series of commands stating
chunks of data to be written to certain addresses. I wrote a small utility to interpret the S-records and write the resulting machine code and data to a file.

Thus we have 5 sections of machine code. For running in the emulators, these were packaged into ROM files containing a simple bootloader program that calls the benchmark function, and then executes a STOP instruction. A minor modification was made to Mandelbrot so that it could be run in the emulators: the calls to system functions that obtain the screen start address were replaced with a hard-coded constant, \$1ffa0000 for the Palm emulator and \$00000800 for the core-only emulator.

For running on the physical m515, an application was written using OnboardC that provides a button for each benchmark, and times each one using the TimGetTicks system function. The machine code for each benchmark is stored as a resource in the application, and is loaded dynamically when the corresponding button is pressed. A screenshot of the application is shown in Figure A.1.

### A.2 Source Code Listings

**Listing A.1:** The Fibonacci program. OnBoardC’s `#pragma segment` directive is used to compile it into a separate code resource.

```c
#pragma segment 50

Int32 fib(Int32 n);

// Fibonacci test
void fibtest(void) {
    fib(32);

    // The inefficiency of this is what makes it a good test!
    Int32 fib(Int32 n) {
        if (n < 2) return n;
        return fib(x - 1) + fib(x - 2);
    }
```

![Figure A.1: Screenshot of the Palm benchmark-runner application, m68test.](Image)
Listing A.2: The Mandelbrot program, another popular demo that involves some more exciting computation. OnBoardC’s `#pragma segment` directive is used to compile it into a separate code resource. Note that when running in the emulator, `disp` is set to the constant $0ffa0000$ instead of calling the system routines.

```c
#pragma segment 51

// Fixed-point Mandelbrot routine
void brot(void) {
    Coord x, y;
    Int32 x0, y0, r, i, r2, i2, d;
    Int16 it;
    // get address to draw to
    UInt8 *disp = (UInt8*) BmpGetBits(WinGetBitmap(WinGetDisplayWindow()));
    // iterate over pixels
    y0 = 0x8000;
    for (y = 0; y < 160; y++) {
        x0 = 0xffff6000;
        for (x = 0; x < 160; x++) {
            // run the Mandelbrot function
            r = 0; i = 0;
            r2 = 0; i2 = 0;
            for (it = 20; it > 0; it--) {
                d = r2 + i2;
                if (d > 0x10000 || d < 0) break;
                // complex multiply: r' = r*r - i*i + x0
                // i' = 2*i*r + y0
                i = (i * r) >> 14;
                i += i + y0;
                r = r2 - i2 + x0;
                // update squares
                r2 = (r * r) >> 14;
                i2 = (i * i) >> 14;
            }
            // set the pixel
            if (it != 0) {
                *disp = 0x00;
            } else {
                *disp = 0xff;
            }
            // move to next
            disp++;
            x0 += 0x199;
        }
        y0 -= 0x199;
    }
}
```
Listing A.3: The no-op loop benchmark. The inner loop contains two \texttt{NOP} instructions rather than only one, which would suffice for the purposes of the test, to prevent the processor entering loop mode, which I felt would give incomparable results.

```assembly
; the number of iterations to run
; reduce by one because DBcc goes to -1 rather than 0
N equ $800000
*-----------------------------------------------------------
* No-op loop test
org $0000
nopTest:
  link A6, #0 ; push stack frame
  movem.l D1-D2, -(A7) ; save registers we use
  move #N, D2 ; D2 is the outer loop counter, start at N
  move #$ffff, D1 ; D1 is the inner loop counter, start at $ffff
  nopLoop:
    nop ; benchmark payload
    nop
    dbra D1, nopLoop ; decrement counter and repeat
    dbra D2, nopLoop ; decrement counter and repeat
  movem.l (A7)+, D1-D2 ; save registers we use
  unlk A6 ; pop stack frame
  rts ; return
```

Listing A.4: The register-bound arithmetic benchmark. In the main loop the counter is simply added to a register. As with Listing A.3 an extra \texttt{NOP} is present to prevent the processor entering loop mode.

```assembly
; the number of iterations to run
; reduce by one because DBcc goes to -1 rather than 0
N equ $800000
*-----------------------------------------------------------
* Register arithmetic loop test
org $0040
addTest:
  link A6, #0 ; push stack frame
  movem.l D1-D3, -(A7) ; save registers we use
  move #N, D2 ; D2 is the outer loop counter, start at N
  moveq #0, D1 ; make sure high word is empty
  move #$ffff, D1 ; D1 is the inner loop counter, start at $ffff
  moveq #0, D3 ; initialise sum to 0
  addLoop:
    add.l D1, D3 ; benchmark payload
    add.l D1, D3
    dbra D1, addLoop ; decrement counter and repeat
    dbra D2, addLoop ; decrement counter and repeat
  movem.l (A7)+, D1-D3 ; save registers we use
  unlk A6 ; pop stack frame
  rts ; return
```
Listing A.5: The memory-bound arithmetic benchmark. Here the counter is added to a local variable held in the stack frame. As with Listing A.3, an extra NOP is present to prevent the processor entering loop mode.

```assembly
; the number of iterations to run
; reduce by one because DBcc goes to -1 rather than 0
N equ $800000

*-----------------------------------------------------------
* Memory loop test
org $0080
memTest:
  link A6, #-4 ; push stack frame (4 bytes)
  movem.l D1-D2/A1, -(A7) ; save registers we use
  lea -4(A6), A1 ; load the address of the local stack variable
  move #N, D2 ; D2 is the outer loop counter, start at N
  moveq #0, D1 ; make sure high word is empty
  move #$ffff, D1 ; D1 is the inner loop counter, start at $ffff
  clr.l (A1) ; initialise sum to 0
  memLoop:
    nop ; benchmark payload
    add.l D1, (A1)
    dbra D1, memLoop ; decrement counter and repeat
    dbra D2, memLoop ; decrement counter and repeat
  movem.l (A7)+, D1-D2/A1 ; save registers we use
  unlk A6 ; pop stack frame
  rts ; return
```

49
Appendix B

Benchmark Measurements

Table B.1: Average execution time in milliseconds measured over three runs of each benchmark program run, in the five test environments.

<table>
<thead>
<tr>
<th>Program</th>
<th>1 Palm</th>
<th>2 CW</th>
<th>3 CA</th>
<th>4 DW</th>
<th>5 DA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci</td>
<td>70703</td>
<td>7992</td>
<td>23724</td>
<td>22627</td>
<td>94348</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>9930</td>
<td>1261</td>
<td>3361</td>
<td>2928</td>
<td>12811</td>
</tr>
<tr>
<td>No-op loop</td>
<td>6777</td>
<td>1077</td>
<td>3251</td>
<td>3316</td>
<td>19138</td>
</tr>
<tr>
<td>Arithmetic loop</td>
<td>7810</td>
<td>1445</td>
<td>3972</td>
<td>3500</td>
<td>20394</td>
</tr>
<tr>
<td>Memory loop</td>
<td>16350</td>
<td>1701</td>
<td>4670</td>
<td>4480</td>
<td>23618</td>
</tr>
</tbody>
</table>

Key

1–5   Test number as defined in the Evaluation (page 35)
C     Run in the core-only emulator
D     Run in the DragonBall/Palm emulator
W     Run on Windows laptop (Windows 64-bit, Intel Core i7, 2.1 GHz)
A     Run on Android smartphone (Android 4.4, Samsung Exynos 5260, 1.3 GHz)
Palm  Physical Palm m515, not emulated
Appendix C

List of Verification Actions

Here I provide a complete listing of every application run and action performed when testing the correctness of my implementation. Each action is accompanied by a screenshot, and an indication of whether it passed (√) or failed (☒).

- Boot screen displayed √
- Digitiser calibration √
- Language selection √
- Warning message √
- Country selection √
- Set the date √
- Set the time √
- Reach end of setup √
- Basic skills opens √
- Tapped ‘Next’ √
- Tapped ‘Next’ √
- Tapped ‘Next’ √
- Tapped ‘Next’ √
- Tapped ‘Next’ √
- Tapped ‘Done’ √
- Tapped ‘Entering Data’ √
- Applications √
General prefs ✓
Network prefs ✓
Owner prefs ✓
Phone prefs ✓
ShortCuts prefs ✓
Web clipping prefs ✓
Launch Security ✓
Set up password ✓
Verify password ✓
Password assigned ✓
Show help ✓
Launch To Do List ✓
Item details ✓
View item note ✓
Delete item note ✓
Delete item ✓
Create new item ✓
Open preferences ✓
Make item personal ✓
Change security ✓
Record is masked ✓
Password prompt ✓
Password entered ✓
Launch Welcome ✓  Tapped anywhere ✓  Tapped anywhere ✓  Completed Digitizer ✓
Appendix D

Project proposal

Computer Science Project Proposal

A Motorola 68000 emulator for Palm OS running on Android

Seb Aebischer, Pembroke College

23 October 2015

Project Originator: Seb Aebischer

Project Supervisor: Ripduman Sohan

Director of Studies: Dr Anil Madhavapeddy

Project Overseers: Jean Bacon and Ross Anderson
Introduction and Project Description

The Motorola 68000 (commonly referred to as m68k) is a CISC architecture designed in 1979 that was used in a wide variety of devices, including arcade games, early Apple computers, and, of most interest to this project, Palm handheld computers.

The broad objective of this project is to write an Android app that will run old Palm applications. More specifically, it is to build an emulator capable of running a virtual Palm OS on device – in particular, the Palm m515. This was among the last of Palm’s devices to use a m68k-based processor, in this case the Dragonball VZ, and ran Palm OS 4.1. Because Palms were completely backwards-compatible, this means it could run applications written for every previous Palm device (excluding ones that relied on external hardware that the m515 lacked). Choosing this model therefore gives my project maximum compatibility.

The proposed project will comprise several parts, divided into a core emulator and a set of extensions to bring it closer to the goal. The core is a software implementation of the m68k architecture that provides a virtual CPU and memory, capable of executing machine code and simulating various other hardware functions present in the Dragonball. This should make up the greater part of the project, regarding implementation, testing, and some degree of optimisation.

After this there are a number of possible extensions that will gradually turn the core emulator into a virtual Palm. The first of these is correctly loading the Palm operating system into the emulator’s memory, followed by emulating the touchscreen hardware, which is important for verifying that everything is working.

There are also half a dozen buttons on a Palm, which would be nice to add, but are not essential. Other Palm peripherals, such as the buzzer, vibrator, and SD card, all have corresponding components on a modern Android phone, and so interfacing each of these provides further extensions.

Starting Point

Many m68k emulators have been written, but there is only one particularly notable for this project. That is the official Palm OS Emulator (POSE), which was distributed as part of the Palm Software Development Kit. This was based on Copilot, an unofficial emulator for the early Palm Pilot devices, which in turn was based on UAE, an emulator for the Commodore Amiga which shares the 68k CPU.

Ideally I would like to start this project from scratch; otherwise it would just be a port of POSE to Android. That would in fact present some difficulties: although POSE was published under GPL and its source code is therefore open, one had to be a registered Palm developer to obtain it. Now that Palm is dead, it is no longer available.

UAE is open source, and although I can’t find the original UAE source code its Windows port, WinUAE, has source code available for download. The source code for Copilot is also available on the author’s website. If necessary I can use either of these as a starting point, or as a reference if I encounter an insurmountable obstacle.
Resources Declaration

For this project I shall use my own laptop (Windows 7, dual-core i7). I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware or software failure. Backups will be done weekly to an external HDD, and I will also regularly copy project files to my MSC home directory. In the event that my laptop fails, I also have a desktop computer (Windows 7, dual-core Pentium) that can be used instead.

For obtaining a copy of the Palm operating system I will need a Palm device and suitable tools. I possess a Palm m515 which will be used for this purpose, in conjunction with the ROM Transfer tool that came with the official Palm SDK. Copying the software for private and educational purposes is regarded as “fair dealing” under Section 29 of the Copyright, Designs and Patents Act (1988), and is not in breach of copyright.

While development will take place on my laptop, I will need an Android device for testing. I plan to use my smartphone, but in the event that it fails a friend’s phone or the Android Emulator could be used.

Project Structure

The project breaks down into the following sections, each of which may be considered a milestone.

1. **Creating the m68k core.** A class will be created for the CPU that stores its state, eg. registers, and provides methods for each of the m68k’s opcodes. There are 85 different opcodes, and their hardware operations must be implemented in code. A fairly minimal Windows console application will be developed that uses the CPU class and allows interaction, such as running an instruction and inspecting register contents.

2. **Adding memory.** A second class will be created for the memory store. This will provide methods for reading and writing data. A further method will then be added to the CPU class that uses the memory and program counter to fetch and execute commands. The console application will be extended to load machine code from a file, run it, and to provide basic disassembly from machine code to mnemonics and arguments, which will help with debugging.

3. **Implementing other Dragonball features.** The Dragonball is an example of a system-on-chip. As such, it contains several subsystems that will require at least a stub implementation, including the following:

   (a) A real-time clock
   (b) An interrupt controller
   (c) Several timers
   (d) Various serial interfaces, including UART and IrDA
(e) Pulse-width modulation (PWM) modules

(f) An LCD controller

The following extensions will bring the project closer to my personal goal:

4. Load Palm OS. The ROM containing Palm OS will be loaded into memory, and efforts will be made to boot it on the emulated device. A working module to display the LCD controller’s output onscreen will be important for this; I define “booting” as displaying the Palm logo, followed by displaying the welcome screen.

5. Move to Android. The emulator classes will be modified as necessary to make them work with the Android NDK, which allows C/C++ code to be run on Android. Java classes will then be written to interface with the emulator core, and a simple user interface will be created.

6. Touchscreen support. Research into the interface between the Dragonball and touchscreen digitiser will be carried out, and code will be written to replicate its functionality.

7. Add hard buttons. Research into the interface between the CPU and buttons will be carried out, and code will be written to replicate their functionality.

8. Pass buzzer commands to phone’s speaker. This has scope for some interesting work: generating a tone of a certain frequency that sounds like an authentic Palm buzzer.

9. Pass vibration commands to phone’s vibrator. Accurate vibration patterns may be limited by Android’s haptic feedback API, but this does not matter too much.

10. SD card support. Classes would be written that allow the virtual Palm to enumerate, read and write files on the Android device’s SD card. This would probably be done by intercepting calls to Palm OS’s Virtual File System API.

Success Criterion

While my personal goal is to be able to run Palm applications inside an Android app, this may not be feasible within the project timeframe, and so the success criteria focus on the core emulator:

1. The emulator must correctly execute each opcode as described in the m68k documentation. This is imperative.

2. Emulation speed is important. The emulated CPU must be at least as fast as its physical counterpart, which, for the Dragonball, is 5.4 MIPS. While this should not be difficult, it should nonetheless be considered a criterion for success.
3. The subsystems of the Dragonball must exist and be emulated sufficiently to allow Palm OS to boot. This criterion is valid only if Milestone 4 (the first extension, loading Palm OS) is reached.

Workplan

Planned starting date is 2015/10/24, after my project proposal is finalised.

1. **Michaelmas weeks 3–6** 2015/10/22–2015/11/11
   
   Read documentation for the m68k architecture, and for the Dragonball VZ processor. Write a couple of basic test programs in m68k assembly. Start implementing opcodes – over 6 weeks, this works out at about 2 per day.

2. **Michaelmas weeks 7–8** 2015/11/12–2015/12/02
   
   Continue implementing opcodes. Start to implement memory module and work on the fetch-decode-execute cycle, using test programs from previous weeks.

3. **Michaelmas vacation** 2015/12/03–2016/01/06
   
   Finish first two milestones. Evaluate its efficiency, and experiment with possible speed-ups. Produce a mock-up Android app demonstrating a user interface.

4. **Lent weeks 0–2** 2016/01/07–2016/01/27
   
   Write progress report. Start working towards third milestone: emulate interrupt controller and LCD controller. These will hopefully allow an interesting demo to be made to accompany the progress report.

5. **Lent weeks 3–5** 2016/01/28–2016/02/17
   
   Work on remaining modules, including clock and timers. Add stub serial interfaces.

6. **Lent weeks 6–8** 2016/02/18–2016/03/09
   
   Buffer period for getting everything working – if on schedule, work on extensions. Start planning dissertation writup in advance.

7. **Easter vacation** 2016/03/10–2016/04/13
   
   Write the main chapters of dissertation. Work on extensions if time permits.

8. **Easter weeks 0–2** 2016/04/14–2016/05/04
   
   Further evaluation and complete dissertation.

9. **Easter week 3** 2016/05/05–2016/05/11
   
   Proof reading before final hand-in.

10. **2016/05/13**
    
    Deadline for hand-in.