Architectural Design Document (ADD)

Computer emulator for digital preservation

Version	**: 1.0**
Author	**: B. Lohman (Tessella Support Services plc.)**
Date	**: 03-03-2006**
Project	**: Emulation project**

Koninklijke Bibliotheek (National Library of the Netherlands)
Nationaal Archief of the Netherlands

I. Revision history

Revision number	Revision date	Author	Summary of changes

II. Related documents

Document name	Date	Author
User Requirements Document (URD)	21-02-2006	B. Lohman
Emulation – a viable preservation strategy	20-06-2005	J.R. van der Hoeven
Reference document for emulation	in progress	J.R. van der Hoeven

III. Table of contents

1 Introduction

1.1 Purpose of this Document

This document describes the proposed prototyping approach for designing the modular emulator. It is based on the user requirements specified in the User Requirements Document [URD] and the use cases specified below.

It will form the basis of the planning of the development phase and will be updated using the results from the various development iterations to come to complete architectural design. At the end of the development the document will be used to write a guide for maintaining the system (System Maintenance Guide, SMG).

It is intended for review by members of the project, notably the system architects of the development team.

1.2 Scope of this Document

This design will follow the Rapid Application Development (RAD) / rapid prototyping lifecycle, and covers the development of the modular emulator via a sequence of fourteen prototypes as described in section "Prototypes" of this document.

Detailed design for many of the components of the prototypes has been delayed until actual development for two reasons: there are many unknowns in design that need to be addressed before deciding a course of action; not all details could be researched in the length of time given for design. These gaps will be filled in using results and outcomes of preceding prototypes, and when research provides answers to details. The complete architectural design will therefore not be complete until the end of the project.

Motives for design decisions made in the project's development phase will be noted here to ensure proper documentation is available for future references.

1.3 Context of this Issue

This document is a work in progress, and detail will be added at later dates when this becomes available. It will continually be updated in a separate document, called the reference document [REF], including all specific know-how about hardware components and their interaction.

1.4 Definition of Terms

ADD	The Architectural Design Document (this document), the high level design document for the entire system.
URD	The User Requirements Document, records the users' requirements for the system.

2 Document Overview and Design Techniques

The overall structure of the document is as follows:

Section 3 contains an overview of the architecture.
Section 4 describes the major system components.
Section 5 lists the prototypes that will be produced in the development phase.
Section 6 lists some use cases for the design stage.
Section 7 finalises with system-wide considerations.

The design methodology used in this project is a combination of RAD / rapid prototyping. The final system will be the result of several iterations of prototypes, expanding and adding complexity with each subsequent iteration. As the project is also a feasibility study, prototypes may implement a solution to a particular issue that seems viable at the time, but at a later stage may hinder further development. It is not unlikely that development will backtrack to resolve these issues and result in different prototypes and / or a different design strategy.

Any such major design decisions will be documented here for further reference.
Use cases have been created to both aid design work and help with test scenarios. At this stage the ADD consists of simple diagrams that have been created to assist in visualising the interaction between the major system components, and so help define their functionality. More detail will be added at a later time, when available and the design requires this.

3 System Overview

3.1 Overview and Context

The diagram of the system is provided below. This shows the proposed system architecture, although not all components stated in this diagram will be developed (including UVM and Module library).

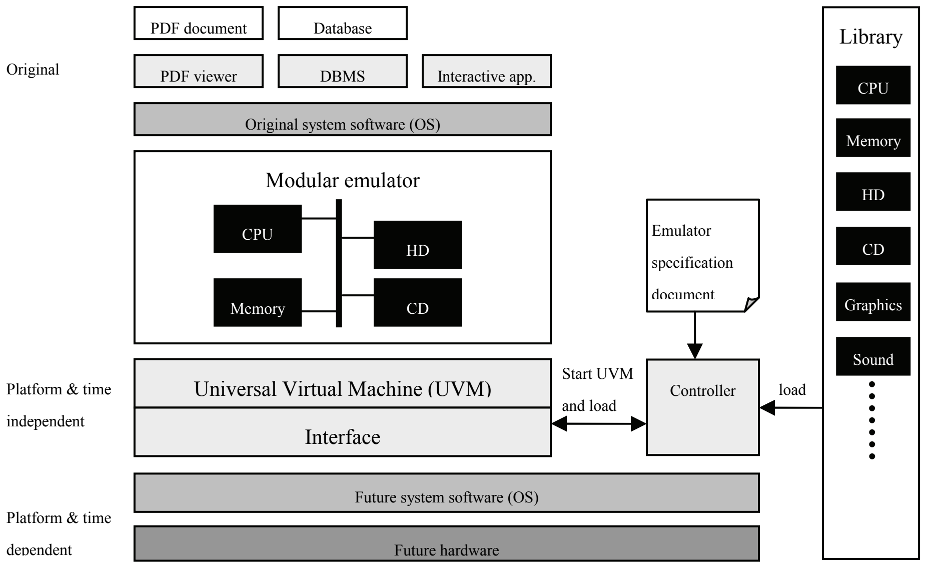

Figure 3.1 Initial design of a modular emulator for digital preservation

A simplified context model diagram, showing the flow of data between the system and its environment across the system boundaries is as follows:

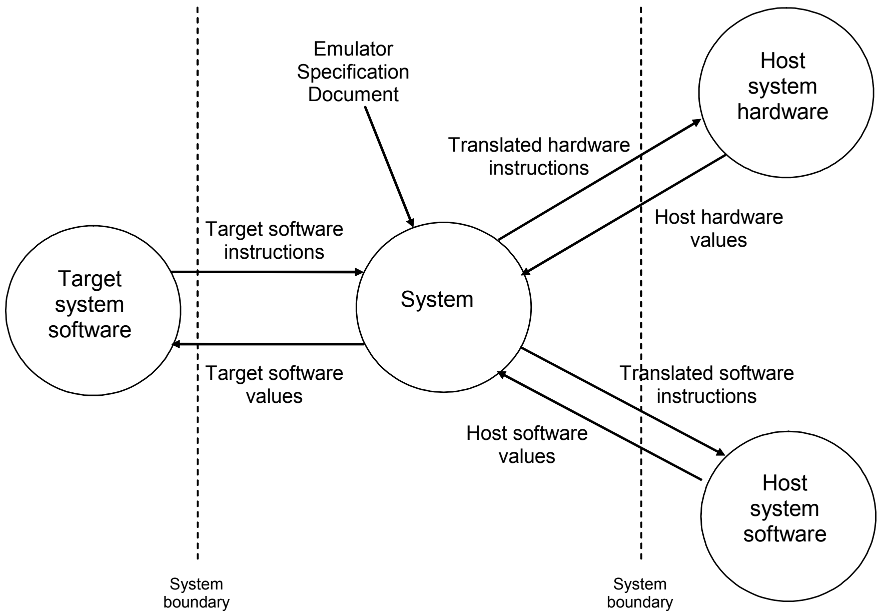

Figure 3.2: simplified context model diagram with major system functionality

The high-level functionality offered by the system is the capability to reproduce the Reference Workstation (RWS) environment as defined in the User Requirements Document (URD). This includes all sub-components, such as graphics, I/O, memory, etc.

3.2 Architecture summary

An overview of the architectural breakdown of the system is given in the following diagram:

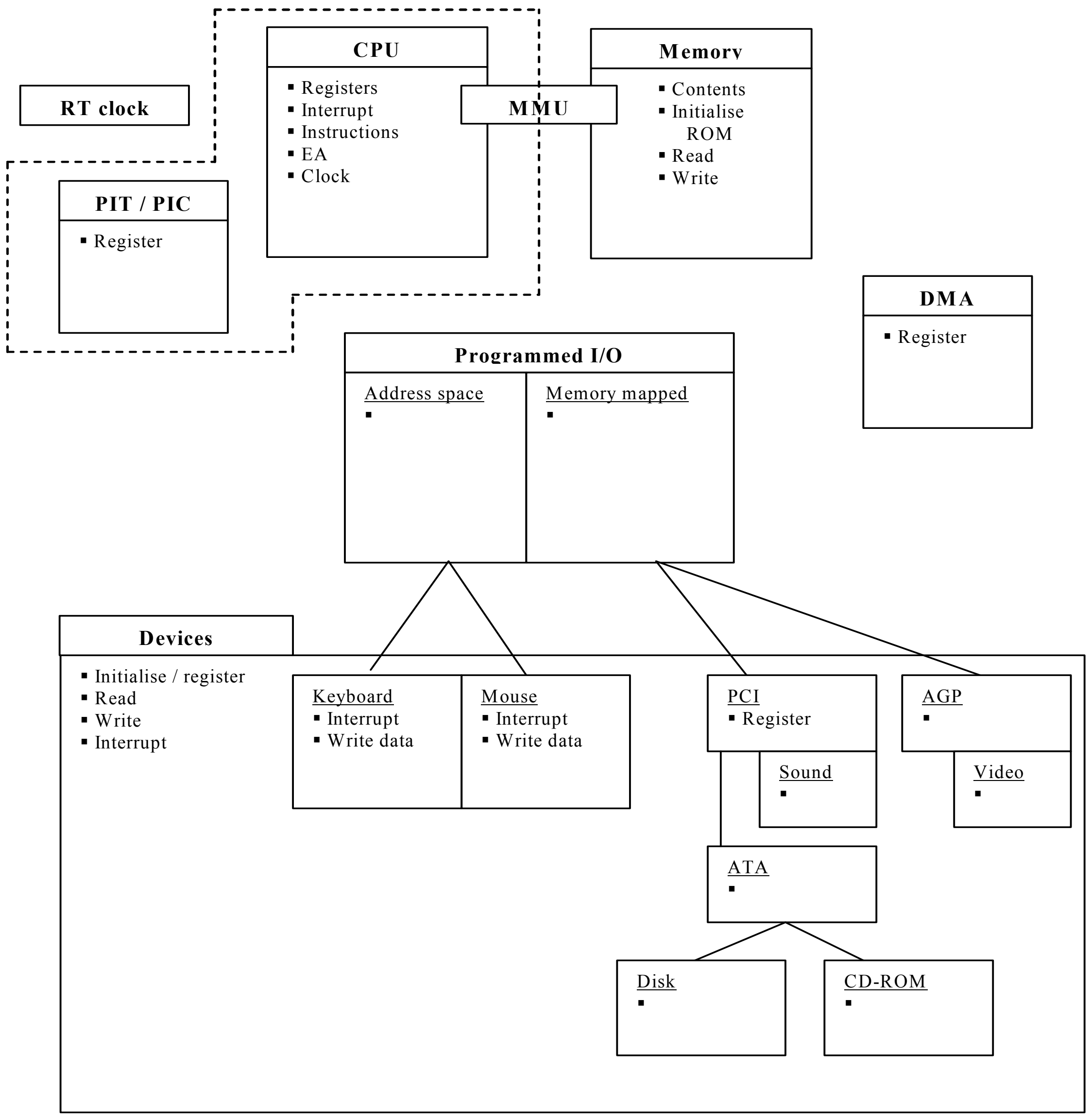

Figure 3.3: architectural breakdown of system

This shows the main components the system consists of, and that will likely need emulating. The complete system will be encapsulated in an emulation environment, not shown in this diagram, which will form the test harness and / or user interface of the system.
Each of these components will be elaborated on in the next section, describing their properties. Once development of the component begins, more detail will be added to each

section. The current version of this document merely provides placeholders with high-level detail.

3.3 Implementation strategy

There are several assumptions that impact the design strategy, which will be stated here for clarity.

Emulation will be implemented at the highest possible logical level
This should improve performance, simplicity and flexibility, amongst others. This also implies that not all hardware will necessarily need to be emulated; as long as the functional behaviour remains the same. However, given the choice of modelling components similar to hardware or logically, with similar performance and development effort, it is likely to be better to model according to the hardware; the model will be easier to understand and any discrepancies due to the emulation should be easier to adapt.
Emulating at a high logical level will also avoid, wherever possible, the need to understand any data or control formats, and avoid the need to modify existing operating systems, BIOS or driver code. Wherever possible, the ‘real’ software will be used for these purposes.

Component modulation is preferred to be kept general rather than RWS specific.
Wherever possible, the components in the RWS will be emulated according to their specifications. However, it is assumed that no applications were designed specifically for the RWS hardware, hence the components need not be emulated exactly. Especially with proprietary components, or due to lack of API information, it will be sufficient to emulate a generic feature of that component. Emulated behaviour should be judged against the "natural range of variability" that is encountered by a preserved digital object in its normal use on a platform, which will have a range of different keyboards, mice, displays, sound chips, processor speeds and features, memory sizes, disks, etc

Not all capabilities of the RWS are relevant for preservation purposes
As stated in the User Requirements Document (URD), priority will be given to those components that are used by the test objects. This defers the need for printing, floppy-disk use, connection of arbitrary devices to ports, network access, sound recording, etc. Other features of the RWS are unlikely to be used by digital objects of interest, e.g., OpenGL graphics.

Modelling of synchronisation mechanisms (such as bus arbitration) may be necessary to give a realistic order of execution of the emulation.
However, it is hoped that this will not be necessary; PC hardware is quite variable and so it would be expected that a range of ordering of hardware activity could result from running a given program, while giving the user a similar experience.

4 System components

4.1 Central Processing Unit (CPU)

Notes:

- Should try to avoid emulating processor pipelining, caching and detailed timing. The modelling of logical behaviour should be independent of these.
- Although likely not to be an issue at present due to low speed performance, may need to emulate instruction cycles, real-time speeds of
- instructions in future to ensure correct running speed.
- Depending on CPU interaction with hardware, ports, mapped memory or special instructions and circuitry may need to be modelled.
- An exception to this is to avoid modelling the exact behaviour of the hardware so that the instructions can result in NOOPs (e.g., not modelling the page table entry caching, INVLPG instruction can be a NOOP).
- It is likely the system bus doesn't need to be emulated.

4.1.1 Attributes

- Registers
    - General purpose
    - Status (including Privilege level, Paging mode)
    - Segment
    - Instruction pointer
- Cycle clock

4.1.2 Operations

- Instructions
- Interrupt handling
- Reset
- Address translation including:
    - lookup of segment descriptors
    - bounds checking
    - segment protection
- EA

4.1.3 Exceptions

- Segmentation fault

4.2 Memory Management Unit (MMU)

Notes:

- Need to determine if paging can be disabled / forced resident. Otherwise need to investigate strategies to minimise (Windows) paging.

4.2.1 Attributes

- Translation Look-aside Buffer (TLB), or other cache.
- invalidation (probably not needed, at least initially)

4.2.2 Operations
- Address translation

4.2.3 Exceptions
- Page fault

4.3 Memory

Notes:
- For performance reasons, this is likely to be implemented as a globally-shared resource, made available equally and efficiently available to processor, memory-mapped I/O devices, DMA devices, etc. Making it as large as possible (within Java limitations) should minimise Windows paging.
- Memory contention should only be emulated as needed. This includes allowing reading from anywhere; multiple writes should be locked out at as fine a granularity as feasible
- Determine access frequency to byte / word / doubleword, etc. This will allow the most efficient access to variable sized chunks of memory
- Access should likely be controlled by accessor functions, encapsulating the memory for segment protection, paging, etc.

4.3.1 Attributes
- Memory locations with values

4.3.2 Operations
- Load. Including bounds checking, (but not alignment checking for x86?)
- Store. Including bounds checking, (but not alignment checking for x86?)

4.4 I/O Ports (programmable I/O memory)

Notes:
- For performance, port "registers" can be emulated as memory locations, irrespective if I/O devices provide physical registers
- Will need to maintain associations of port addresses to emulated devices.

4.4.1 Attributes
- Port address space (with buffered values?)
- Association of addresses with devices

4.4.2 Operations
- Read
- Write
- Signal status changes to attached devices

4.5 Advanced Programmable Interrupt Controller (APIC)

Notes:

- Functions should only be emulated if necessary

4.5.1 Attributes
- APIC registers - mapped to IO ports
- Interrupt priorities
- Interrupt mask

4.5.2 Operations
- Interrupt handling - this physically includes a fan-in of interrupts from attached devices that probably doesn't need to be modelled explicitly
- Interrupt priorities pre-emption
- Interrupt masking (?)

4.6 Real-time clock

Notes:
- Device traditionally on IRQ 8

4.6.1 Attributes
-

4.6.2 Operations
- Set time
- Return time

4.7 Direct Memory Access (DMA) Controller

Notes:
- For simplicity, should avoid emulating DMA Controller. Instead, control ports can be used to initialize a transfer; emulate interrupt-generation when transfer is done / on error
- Emulate latency and transfer speed, with option to run full-speed.

DMAC functionality seems greatly reduced in modern PCs, and logically the functionality can belong with the DMA device.

Two modes:
- LPC (channels 0-7)
- PC/PCI

4.7.1 Attributes
- Multi-channel

4.7.2 Operations
- Arbitrate bus access (probably not needed in this form; would need memory access synchronisation for a multithreaded emulator)
- Generate IRQs

4.8 PCI Bus

Notes:

- Likely able to avoid emulation of component, but may need to emulate any control ports used to control device transfers

4.8.1 Attributes
- Attached devices (with device and function numbers)

4.8.2 Operations
- Probe for devices

4.9 Graphics card

Notes:
- As understood at present, display is character-mapped and/or bitmapped. Need to check if Java can model necessary resolution, colour-depth, etc.
- Defer low-priority transformations (e.g. OpenGL, vector/shape/texture models initially
- Look into possibility of separate threads for performance

4.9.1 Attributes
- Port(s)
- Memory location(s)
- IRQ(s)
- Video mode
- Frame buffer

4.9.2 Operations
- Initialise/register at startup
- Handle status change
- Read data from port or memory
- Write data to port or memory
- Raise interrupt
- Display frame buffer

4.10 Keyboard device

4.10.1 Attributes
- Port(s)
- Memory location(s)
- IRQ(s)
- Shift states?

4.10.2 Operations
- Initialise/register at startup
- Handle status change
- Read data from port or memory
- Write data to port or memory
- Raise interrupt
- Send keyboard event

4.11 Mouse device

4.11.1 Attributes
- Port(s)
- Memory location(s)
- IRQ(s)
- Button states?

4.11.2 Operations
- Initialise/register at startup
- Handle status change
- Read data from port or memory
- Write data to port or memory
- Raise interrupt
- Send motion event
- Send button event
- Send wheel event

4.12 ATA Controller

Notes:
- Assume that transfer will be by DMA; it may be possible to disable this for a real disk.

4.12.1 Attributes
- Port(s)
- Memory location(s)
- IRQ(s)
- Attached disks (geometry...)

4.12.2 Operations
- Handle status change
- Read data from port or memory
- Write data to port or memory
- Raise interrupt
- Read block range (via DMA)
- Write block range (via DMA)

4.13 Hard disk device

Notes:
- Uses ATAPI interface (SCSI commands over ATA; data transfer over ATA PIO or DMA protocol)
- Assume that transfer will be by DMA; it may be possible to disable this for a real disk?

4.13.1 Attributes
- Port(s)
- Memory location(s)
- IRQ(s)

4.13.2 Operations
- Initialise/register at startup
- Handle status change
- Read data from port or memory

- Write data to port or memory
- Raise interrupt
- Read block range (via DMA)
- Write block range (via DMA)

4.14 CD-ROM devices

Notes:

- Uses ATAPI interface (SCSI commands over ATA; data transfer over ATA PIO or DMA protocol)
- Assume that transfer will be by DMA; it may be possible to disable this for a real disk?

4.14.1 Attributes

- Port(s)
- Memory location(s)
- IRQ(s)

4.14.2 Operations

- Initialise/register at startup
- Handle status change
- Read data from port or memory
- Write data to port or memory
- Raise interrupt
- Read block range (via DMA)

4.15 Sound device

Notes:

- Need to check Java sound AC'97 ability
- Look into possibility of separate thread for performance. Long-lived thread vs. throw-away sound threads

4.15.1 Attributes

- Port(s)
- Memory location(s)
- IRQ(s)
- Mode; mono/stereo, sampling rate

4.15.2 Operations

- Initialise/register at startup
- Handle status change
- Read data from port or memory
- Write data to port or memory
- Raise interrupt

4.16 Emulation Environment

4.16.1 Attributes

-

4.16.2 Operations

- Configure emulation
- Configure emulated devices
- Load BIOSes
- Initiate emulation
- Transfer control to emulator

5 Prototypes

The design of the emulator is based around a sequence of prototypes, where each subsequent iteration includes more functionality. The final product should provide the functionality of the modular emulator as described in URD.
This approach was chosen to facilitate design, allow simple implementation to show where bottlenecks lie, and defer research for in-depth knowledge until necessary during component creation. The sequence is as follows:

Table 5.1: prototypes

Sequence	Prototype	Details
1.	Registers, memory model, test harness	• 8 general purpose, 6 segment, EFLAGS, EIP • Test segmented memory access
2.	Subset of instruction set/address modes, simple interrupts, clock (?)	• Instruction decoding • Real-mode Effective Address (EA) computation • Simple instructions: Load, Store, Store Immediate, Increment, Conditional-Jump, Loop, subroutine Call & Return, Return from Interrupt (IRET), Interrupt Flag instructions, LEA (Load Effective Address), HALT • Respond to interrupts: Save EFLAGS, EIP, CS registers and optionalerror code on stack • Log/report attempts to execute instructions or modes that are not implemented yet
3.	Simple Emulation Environment	• Show registers & requested memory locations • Allow typing in register/memory data or loading from files • Allow typing in machine language/ASM or loading from files • Allow running emulator at specified start location • Allow single-stepping • Allow forcing an interrupt request (IRQ)
4.	Minimal machine-language test programs	• Store, Increment, Load • Store Immediate into Segment Register, Store, Increment, Load • Loop • Call in response to interrupt • Return from Call • Use a debugger to try same code on real h/w or

		validated simulator • Evaluate execution speed
5.	Simplified memory-mapped character display in a Java window Simplified AGP (PCI) for memory-mapped I/O Bit-mapped Java I/O display card emulation, test harness Simplified sound card output, test harness	• Hand load characters into memory-mapped area and show that they appear on the display • Do bus management (if any) for memory-mapped I/O control • Hand load VGA bit-mapped graphics into memory-mapped area, use Java graphics to display the result in a window • Hand load sound-generation data into memory-mapped area and use Java sound to play resulting sound
6.	Minimal graphics/sound machine-language test programs	• Add CPU instructions as needed to run programs that move graphics/sound data to memory mapped areas and cause resulting display/sound output (no I/O port control yet) • Use a debugger to try same code on real h/w or validated simulator • Evaluate execution speed
7.	Port address space, programmed I/O	• Trap I/O ports to call back-end Java device emulators • Keyboard & mouse I/O, test harness • Map I/O port addresses to devices • Implement IN and OUT instructions • Perform keyboard input and mouse sensing • Show mouse-controlled cursor on bit-mapped display • Return mouse-click location at bit-mapped display position • Set interrupt priorities on APIC • Trap and log unrecognized I/O calls
8.	Disk I/O (simple ATA / DMA)	• Cycle clock, instruction cycle emulation • Real-time clock • Set up parameters for DMA transfer via

		Programmed I/O • Access disk image (stored on real disk) by track/sector • Transfer data between emulated disk and main memory • Signal completion of transfer via IRQ • Implement clock functionality if needed for DMA emulation (e.g., to emulate CPU cycle-stealing)
9.	Expand instruction set, emulation environment	• Add additional instructions, modes needed for BIOS • Extend emulation environment to read, test files of machine code • Run tests/benchmarks • Evaluate Java speed, graphics, sound, disk access
10.	Implement BIOS	• Including simple I/O, disk-access, boot • Use Bochs / QEMU code if appropriate
11.	Boot emulator	• Make emulation environment load low-memory, BIOS, and initialize VRWS
12.	Boot LILO, DOS and/or minimal Linux in Real Mode	•
13.	Expand emulator as needed for Windows	• Emulate transparent MMU, TLB, etc. to avoid page-faults • Add virtual addressing and other modes • Build test harness and page tables by hand, to test • Run test programs that cause paging
14.	Boot Windows	• Redesign as needed and iterate until Windows runs

6 Design Scenarios

Listed below are a number of simple use cases that served as a basis for the architectural design, and can be reused during the testing phase.

Table 6.1: Use cases for emulation

Use Case	Details
CPU load/store (including MMU) [instruction decode, including EA comp]	Real-mode: • call memory Other modes: • use MMU, page-fault interrupt, PIC [services page fault via DMA, uses programmed I/O to set up DMA] • call memory
CPU service interrupt [CPU addresses PIC port to set interrupt priority]	• Disable interrupts • Call handler; re-enable interrupts; return • Priority / pre-emption, etc.
Get keyboard character	• Keyboard checks I/O port memory status • Keyboard writes input character to I/O port memory • Keyboard raises IRQ [Interrupt caught by PIC, which passes it on to CPU] • CPU responds to interrupt [notify PIC that interrupt is handled] • Read I/O memory • CPU may modify I/O port memory status word
Display image	• Copy bit-mapped screen image to display memory Hypothesis: There exists a standard, simple graphics output format used by Windows and applications for any card; this MAY be simple bit-mapped, but in any case should be well documented
Read hard disk	• CPU uses DMA/disk Controller I/O port to initiate disk read of specified tracks/sectors into specified memory area • DMA/disk Controller performs disk read into memory • DMA/disk Controller may modify I/O port status word • DMA/disk Controller raises interrupt

	[Interrupt caught by PIC, which passes it on to CPU] • CPU responds to interrupt • CPU may modify I/O port status word
Write hard disk	• CPU uses DMA/disk Controller I/O port to initiate disk write of specified tracks/sectors from specified memory area • DMA/disk Controller performs disk write from memory • DMA/disk Controller may modify I/O port status word • DMA/disk Controller raises interrupt [Interrupt caught by PIC, which passes it on to CPU] • CPU responds to interrupt
Initialize VRWS	• Initialize registers, memory (to non-zero pattern?), log files, etc. • Emulator environment loads stored low-memory image with BIOS, interrupt-vector, etc. • Emulator environment reads file to load device IDs and configurations for Programmed I/O, PCI, etc., initializes I/O port associations with emulated devices
Boot from disk image	• Perform INT 19 to jump to the BIOS boot entry • Execute standard BIOS boot code • Boot code searches for disk, loads Master Boot Record (MBR) (Track 0, Sector 1) at 0000:7C00 and jumps there • Stored boot sector image is executed, which tries to load OS kernel loader
Output sound	• CPU uses the sound card I/O port to set up sound output parameters • CPU copies sound stream to sound memory • Use I/O port to tell sound card to start playing sound • DMA Controller / sound chip reads sound data from mapped memory • Sound card plays sound • DMA Controller / sound chip may modify I/O port status word • DMA Controller/sound chip raises interrupt [Interrupt caught by PIC, which passes it on to CPU]

	• CPU responds to interrupt • CPU may modify I/O port status word
Read/Write from CD	This use case is identical to “Read hard disk”

7 System-wide considerations

7.1 Automated Debugging, Testing and Diagnostics

The major methods of classes will have their own automatic self-checking tests using the JUnit framework to enable frequent, extensive unit tests to be run throughout development. This should include an approach for tracing the emulator's operation. The overhead of logging must be evaluated and perhaps a means of conditionally including logging found.

7.2 Speed

This is expected to be a critical issue; initial performance must be enough to demonstrate the ultimate utility of emulation for preservation.
During development, performance will be measured and improved using profiler tools. The use of multithreading and/or multicore/multiprocessor systems will be examined, as will the influence of different JVMs.
It has been noted that performance will improve over time as host platforms get faster, but this cannot be relied on at present.

7.3 Third-Party Software

Existing open source emulators exist that have addressed the problems faced in this project. Although the exact implementation might not be the same, code reuse is a possibility. Of particular interest are Bochs and QEMU, both which are release under the GNU Lesser General Public License (LGPL), which allows their source code to be integrated into this project.

7.4 Development Environment

Following in the tradition of the open source community, open source tools will be used for the development process; these are available at no cost. Added benefit is that most run in both Windows and Linux, thus providing the ability to develop in both these environments.
The following tools will be used:

- Sun JDK 1.5.x or higher
- Eclipse (integrated development environment for Java)
- JUnit (automated testing environment for Java)
- CVS (“Concurrent Versions System”). A CVS server will be set up and used at the Nationaal Archief of the Netherlands. When the results will be released, the CVS will be switched to SourceForge.