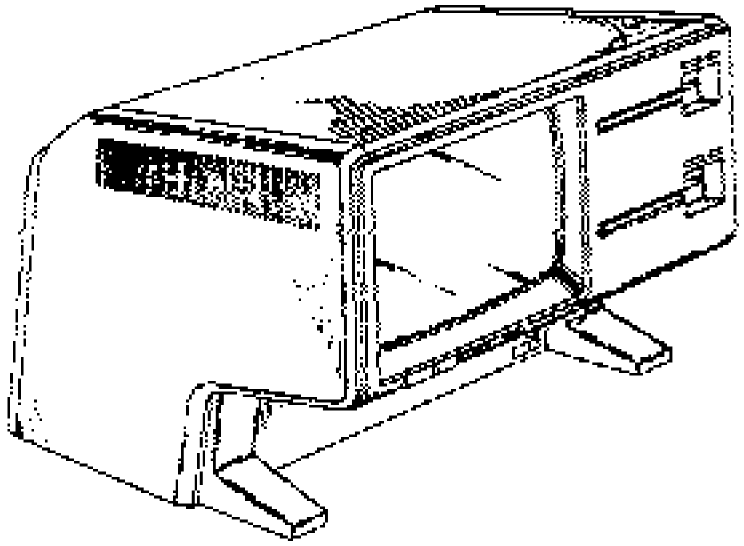
A FEW IDEAS ABOUT THE DESIGN AND IMPLEMENTATION OF A SOFTWARE EMULATOR FOR THE APPLE LISA COMPUTER



941 Calle Mejia # 1006, Santa Fe, NM 87501 USA

home phone: (505) 820-0358
email: 71533.606@compuserve.com

Revision 4 -- 06 July 1998

TABLE OF CONTENTS

SECTION 0 -- DOCUMENT MODIFICATION HISTORY

Revision 0 / 07 June 1998

Created

Revision 1 / 16 June 1998

3: Added interrupt and reset switch info, added screen dimensions and pixel aspect ratio
4: Added info on LisaTest, Office System, MacWorks
6: Added more memory info, added info about emulator hardware abstraction layer
9: Added debugger commands
12: Created new section #12 FUN STUFF
13: Changed section # from 12 to 13, added ImageWriter printer info

Revision 2 / 18 June 1998

0: Reformatted, changed "Rev" to "Revision".
3: Added more info about the screen.
6: Changed "assessed" to "accessed", added comments about rolling your own 68k CPU emulator, added more info about the MMU registers (e.g. changed the TMMURegisters data structure).
9: Changed "timers" to "timer" in Set Timing Bucket command, reformatted all debugger command descriptions so the abbreviation appears first followed by "/" followed by the abbreviation spelled-out, added examples to all debugger commands, abbreviated commands are now the only commands accepted by the debugger, LP command address parameter is now mandatory, removed the Clear All Timing Buckets command since Clear Timing Bucket now does this, changed Reset Timing Bucket to also reset all the buckets, for command TM changed "MEMORY TRAP" to "MEMORY TRAP - READ" or "MEMORY TRAP - WRITE", added version info commands, added PICT and PICTP commands, changed MMU commands so that origin, limit, and control components of a MMU register are displayed and set separately.
11: Changed "Windows machine" to "Windows machines", mentioned 2MB memory was also supported.

Revision 3 / 03 July 1998

all: Used past tense verbs for all Lisa descriptions.
6: Improved the logical to physical address diagram, added big/little-endian info.
7: Mentioned my DTCMacDiskUtility program.
9: Corrected DM command example, added detailed snapshot file format for the DEM command, made file-name arg optional in LOG command, added commands TMV and TMP, changed DTM command to show value and pattern info.
13: Added Wakerly's 68000 book, added Lisa Owner's Guide, improved ImageWriter printer info, added Linzmayer's Apple history book.

Revision 4 / 05 July 1998

all: Changed "chapter" to "section", spell checked everything.
3: Added a picture of a Lisa Office System screen, added listing of the 7 LOS tools, now printed on a laser printer instead of my ImageWriter due to the screen image which needs a good printer.
9: Improved the memory snapshot file format.
13: Added info about the Workshop utility programs.

SECTION 1 -- ABSTRACT

This document describes a few ideas I have about the design and implementation of a software emulation program that would emulate the Apple Lisa computer. This emulator should be strictly software-based and not hardware-based (i.e. no special physical devices should exist to make this emulator work).

SECTION 2 -- ABOUT THE AUTHOR

David Craig has been involved with the Lisa computer since 1984 when David used a Lisa to cross-develop Macintosh applications. David started with a Lisa 2 that was upgraded from a Lisa 1 and used version 2 of the Lisa Workshop. David later obtained a Lisa 2/10 with Workshop version 3. Since those days David has become very interested in the technology behind the Lisa and has tried to collect as much Lisa technical information as possible (this collection contains around 350 documents). David owns today two Lisas and the majority of the software ever made for the Lisa. This includes the Lisa Office System and the Lisa Workshop development environment. David also has all the documentation for these programs. David can copy whatever a person wants for 10 cents per page plus postage. See the REFERENCES section for the details behind the Lisa documents which should be important for emulator writers.

SECTION 3 -- LISA BACKGROUND

The Lisa was a micro-computer system sold by Apple Computer between 1983 and 1985. The Lisa was one of the first personal computers to sport a graphical user interface (GUI) which was controlled by a mouse. The Lisa's claim to fame was that it introduced the GUI to the micro industry. The Lisa's user interface was also rather radical and was based on a document-centric architecture. This architecture focused on documents, not applications. For example, the Lisa user never ran programs, instead the user used stationery pads to create documents which when opened (via clicking with the mouse) the appropriate Lisa program was run to process the document. Apple developed an extensive set of user-oriented software for this machine which was very integrated (the Lisa name per Apple represented Local Integrated Software Architecture). There were 7 tools in this suite:

o	LisaWrite	--	Word processor
o	LisaDraw	--	Object-oriented graphics editor
o	LisaCalc	--	Spreadsheet
o	LisaGraph	--	Simple graphics chart maker
o	LisaProject	--	Project scheduling manager
o	LisaList	--	Simple database manager
o	LisaTerminal	--	Terminal program

The Lisa commercially did not fair well but it did cause Apple to produce another GUI machine in 1984 named the Macintosh which did much better commercially. This GUI was adopted by other companies such as Microsoft and Sun for their desktop-based operating systems.

The Lisa's hardware consisted of a keyboard, a single-button mouse, a 12" B/W bitmapped screen, 2 internally connected 860K Twiggy floppy disk drives, 1 externally connected 5MB ProFile hard disk, a clock chip, and 1MB of main memory (1MB was standard, but could be increased to 2MB total with 3rd party memory cards). The later Lisa 2 computer sported a single 400K Sony disk drive and an internal 10MB Widget hard disk. The back of the Lisa contained 3 peripheral expansion card slots, 1 parallel port, 2 serial ports, and a mouse connector. The expansion slots supported special cards (e.g. a 2 port parallel card which was used to communicate with ProFile hard disks, a 4 port serial card mainly for Xenix terminal connections, an ethernet communication card made by 3Com, and a manufacturing test card used only by Apple). The heart of the Lisa was the Motorola 68000 CPU which ran at 5MHz.

The Lisa's screen had the following characteristics:

o 720 pixels wide by 364 pixels high, very clear, no fuzziness

o Each pixel was represented by a memory bit. There were 262,080 bits, 32,760 bytes.

o Each pixel had an aspect ratio of 2/3. This means there were 3 pixels in the horizontal direction for every 2 pixels in the vertical direction. A rectangle 30 pixels wide by 20 tall looked square on the screen. This aspect ratio was chosen to make text appear better.

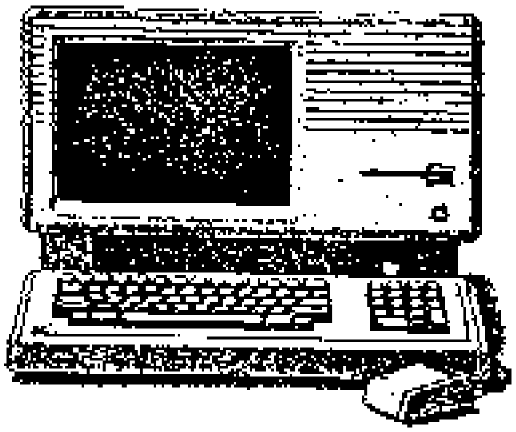
o There were 90 pixels per inch horizontally, 60 pixels per inch vertically.

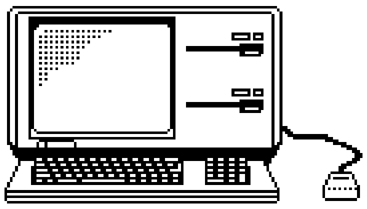
o Black pixel bits were set to 1, white pixel bits were set to 0.

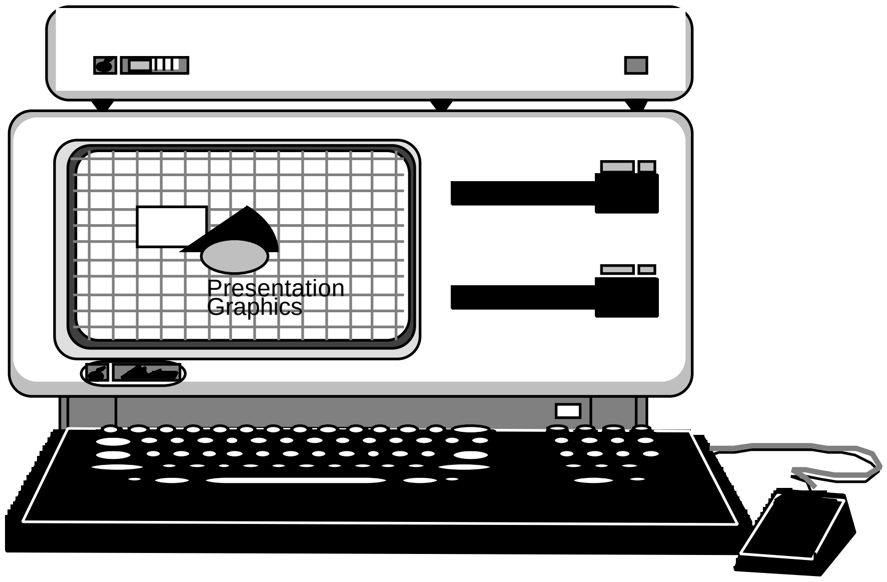
Also on the back of the Lisa was an Interrupt switch and a Reset switch. The interrupt switch when pressed interrupted the Lisa's operation and would activate the Lisa's low-level debugger (LisaBug) if installed. The reset switch would reset the Lisa completely causing it to re-boot.

The Lisa also contained on the front a keyboard connector and a soft power-ON/OFF switch which when ON glowed.

Here are some pictures of the Lisa 1 and Lisa 2 computers. Another Lisa 1 picture (from Apple's Lisa design patent) appears on the first page of this paper:

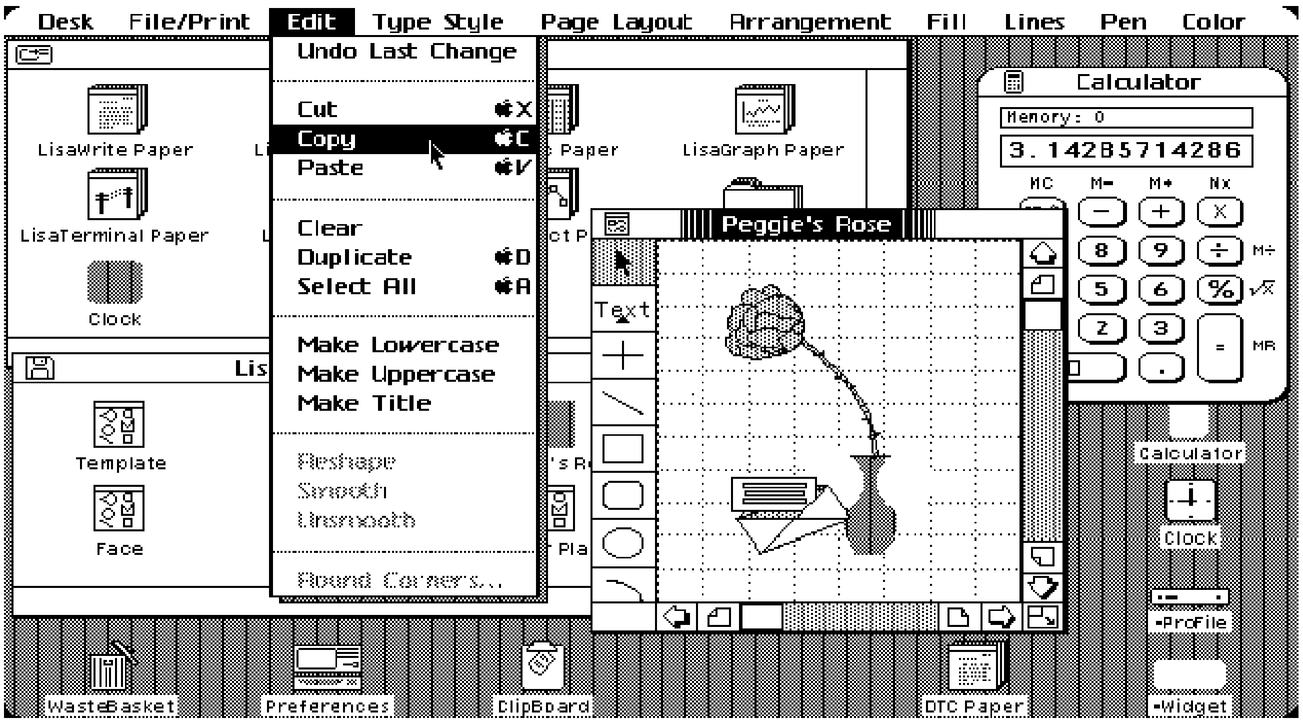

Lisa 2 Computer


Lisa 1 Computer



Lisa 1 Computer

Here is a sample screen picture from my Lisa running the last Lisa Office System, version 3.0 (a.k.a. Lisa 7/7):



SECTION 4 -- GOALS

The goals of a Lisa emulator should be:

- o Faithful emulation of the Lisa's hardware and software.
- o Support for Lisa disks.
- o Support for printing.
- o Faithful execution speed.

Users of the emulator should be able to sit down in front of the emulated Lisa and think they were running a real Lisa. For example, when the Lisa emulator user starts the Lisa the user would see the boot ROM diagnostic icons that the real Lisa displays. The Lisa emulator should be able to boot Lisa disks and run the programs on those disks. For example, users should be able to "insert" the LisaTest disk and have the Lisa run this diagnostics program. Or the Lisa emulator should be able to run the Lisa Office System software. I think that if a Lisa emulator can run the LisaTest program and all of this program's tests pass then the emulator will be correctly implemented.

One area of emulation that could be rather neat would be for the Lisa emulator to boot and run the Apple MacWorks emulation disk (I recommend MacWorks 3.0, the last Apple produced). Then you would have a host machine emulating the Lisa which would be emulating the Macintosh -- two emulators in one!

The emulation speed should also be at least as fast as a real Lisa. Given that the Lisa's CPU ran at 5MHz I don't think this would be a problem and that the emulator may even need to have a user-controlled speed setting for when the emulation was too fast.

I recommend highly that a Lisa emulator writer have access to a working Lisa that is running Lisa software. I recommend that the Lisa run the Lisa Office System and the Lisa Workshop. Don't use a Lisa running the MacWorks software since you will then basically have a Macintosh and not a Lisa.

SECTION 5 -- USER INTERFACE

The Lisa emulator should appear to the user in either a single window on the screen or the whole screen of the emulation machine should be devoted to the Lisa. The single window would represent the complete Lisa screen. Within this window the Lisa software would create its own windows as the real Lisa did. I recommend the single window approach for various reasons:

o   The single window would not interfere with the user's other machine activities. If the user decides to do other machine tasks the user could just access these tasks as normal (e.g. run a word processor).

o   The bottom of the window could contain additional emulator-specific commands. These "commands" which could be icons representing specific functions would be used by the Lisa user for special purposes. For example, there could be the following icons or functions in this area:

+   Icon representing the Lisa's power-on/off button. When the Lisa user wanted to turn "on" the Lisa the user would press this button and it would "light" just like the real button does on the Lisa. When the user wanted to turn "off" the Lisa the user would press this button and the Lisa emulator would turn off.

+   Icon to mount a disk. This icon when pressed would ask the user to select a Lisa disk image file for mounting. Once the image file was selected the emulator would then inform the Lisa of the new disk and the Lisa would mount the disk as normal. To unmount/eject a Lisa disk the user would need to select the disk in the Lisa Desktop Manager and use the DM's Eject menu command.

+   Icons representing all the mounted disks. For example, when the user mounted a disk an image of the disk would appear in this area. The icon would contain below it the name of the Lisa disk volume (the emulator may need to know where to get this volume name from a Lisa disk image -- or use a generic name for the disk, e.g. "Disk # 2").

+   User preferences icon. This icon's function is to allow the user to specify certain emulation preferences. These could include things like the speed of the emulator. I recommend that the emulator speed be set to something reasonable originally but also allow the user to either slow down or speed up the emulation to match their speed tastes. I have a feeling that the emulator will for the most part be too fast for most users and they will want to slow it down so it acts like a real Lisa.

+   Reset and Interrupt buttons. These should act just like the Lisa's buttons. The reset would restart the emulation (not the emulator). The interrupt should either activate LisaBug or cause the Lisa software to display an "interruption has occurred" dialog.

A second window could also appear in the emulator. This window would be used for the Emulator Debugger and would normally not be visible to the user. Advanced emulation users or emulator programmers should be able to activate and access this window. See the section below for details behind this debugger window.

SECTION 6 -- EMULATOR SOFTWARE ARCHITECTURE

The Lisa emulator program should be divided into three sections:

1)	Emulator kernel	-- handles all the Lisa hardware emulation.
2)	Emulator debugger	-- allows a behind-the-scenes emulator view.
3)	Emulator utilities	-- includes things like emulator preferences.

The emulator kernel would contain all of the programming necessary for the emulator to emulate a Lisa. This programming would include code for the following:

LOW-LEVEL KERNEL AREAS:

- 68000 instruction parsing, analysis, execution and timing
- MMU register reading, writing, and using
- VIA1 and VIA2 parallel port controllers
- SCC serial port controller
- COPS keyboard and mouse controller
- Screen drawing to both the main and alternate screens
- Screen Contrast control (emulator may not support this)
- Speaker Volume and sound tone
- Floppy and hard disk control
- Lisa Serial number reading
- IRQ interrupt and switch handler
- Reset interrupt and switch handler

Emulation of the 68000 CPU instruction set is a must. The Lisa emulator should contain a complete 68K CPU emulator which is either written by the Lisa emulator authors or uses source code for an existing emulator (e.g. I've heard that several 68000 emulator source packages are available on the internet). This emulator must work closely with the Lisa Emulator Debugger (see the DEBUGGER section).

The 68000 is a big-endian CPU so if your host machine is little-endian the emulator will need to handle the 68000's 2 and 4 byte register and memory contents appropriately.

A key aspect of this kernel is the complete emulation of the Lisa's hardware operations. Also present here should be complete control over all aspects of the Lisa including all memory accesses. For example, when a memory access is made the 68000 instruction analysis section of the emulator should determine if the access is for regular memory or memory-mapped I/O memory. If for regular memory then MMU logic needs to come into play. If for memory-mapped I/O memory then the appropriate hardware action should take place (e.g. reading the Lisa's speaker volume setting).

Emulator writers must also have a working Lisa that runs the Lisa OS, not other OSs such as MacWorks. The reasons for this are so the emulator writers can first see exactly how a real Lisa looks while it is running and can peek at the internal operations via techniques such object file disassemblies, LisaBug disassemblies and MMU register dumps.

Emulating the Lisa mouse may be a problem. I recommend that the emulator not use the host machine's mouse pointer bitmap for the Lisa mouse. Instead, the emulator should track the host machine mouse _and_ then draw the Lisa mouse image on the Lisa screen as appropriate. This would require the emulator to handle the Lisa mouse's low-level software implementation and data structures. The Lisa boot ROM has a great listing of the mouse routines which I believe are the only mouse routines present in the Lisa (i.e. the Lisa hardware library (residing in Lisa file SYSTEM.LLD) does not have its own mouse routines but uses the ROM mouse routines).

For memory handling I recommend that there exist several memory-based data structures which represent the Lisa's memory:

- o 68000 CPU registers
- o Main memory (1MB or 2MB)
- o Boot ROM (16KB)
- o MMU register set
- o Memory-mapped I/O

68000 CPU registers should be supported with the following data structure (written as a Pascal record, see a 68000 book for the details):

```
T68000Registers = RECORD
      D     : array [0..7] of longint; { data registers }
      A     : array [0..7] of longint; { address registers (A7 = user stack ptr) }
      SS    : longint;                 { supervisor mode stack pointer }
      PC    : longint;                 { program counter (0-FFFFFF) }
      STATUS : packed record
                  trace             : boolean;
                  b14               : boolean;
                  supervisor        : boolean;
                  b12               : boolean;
                  b11               : boolean;
                  interrupt_mask    : 0..7;
                  b7                : boolean;
                  b6                : boolean;
                  b5                : boolean;
                  extend            : boolean;
                  negative          : boolean;
                  zero              : boolean;
                  overflow          : boolean;
                  carry             : boolean;
            end;
END;
```

Note that only the low 3 bytes of the PC are interpreted by the Lisa since the 68000 only supported 16MB of memory. The high byte is ignored.

Main memory emulation support should consist of a single 1MB or 2MB memory data structure that is byte-addressable. Note that the Lisa's screen memory was just part of the main machine memory. There was no special memory for the screen. Screen memory occupied 32KB. There can also be a secondary screen (called the alternate console) which also resided in the main memory. If the emulator host machine cannot accommodate 1MB (or 2MB) of real memory then this memory could be handled in a virtual fashion by the emulator.

Boot ROM memory should consist of a 16KB memory data structure that should contain a copy of this ROM and be read-only (the Lisa MMU may already protect this section of memory, but I recommend that the emulator monitor this area for writes just in case).

The Lisa's memory organization was based on a segmented architecture which contained a physical memory and hardware (called the MMU) that managed all accesses to the physical memory. The Memory Management Unit (MMU) hardware's purpose was to map logical memory addresses to physical memory addresses. As far as Lisa applications were concerned all of their addresses were logical and could span 16MBs even though the Lisa's physical memory was much smaller than 16MBs. The MMU consisted of a set of registers that determined the logical-to-physical mapping. The Lisa's boot ROM and OS maintained the contents of the MMU registers. See the Lisa hardware manual for detailed MMU information.

Here's an overview of the MMU registers and how they were used to map logical addresses to physical addresses. MMU registers should be supported by the following data structure which represents 4 sets of MMU registers with each set consisting of 128 24-bit registers:

```
TMMURegisters = array [0..3,0..127] of
   RECORD
      segment_origin: 12 bits  { page physical start address (512 byte page-based) }
      segment_limit:   8 bits  { page physical size }
      segment_control: 4 bites { page access control }
   END;
```

There were 4 sets of MMU registers to support fast context switching between Lisa software processes.

A logical address contained 24 bits with a range of $000000 to $FFFFFF (0 to 16MB-1). A logical address was composed of 3 components (as far as the Lisa's hardware was concerned, the Lisa programmer saw only logical linear addresses), "$" represents a hexadecimal number:

o	Segment number:	Bits 23-17	Width 7	Values 0-127/$7F
o	Logical block number:	Bits 16-9	Width 8	Values 0-255/$FF
o	Logical displacement:	Bits 8-0	Width 9	Values 0-511/$1FF

Here's a diagram of how 24 bit wide logical addresses were converted by the MMU into 21 bit wide physical addresses (the numbers outside of a box represent the number of bits in the closest data path):

```
                            LOGICAL ADDRESS (24 BITS)
+-------------------+------------------------+------------------------------+
| 23   SEGMENT   17 | 16   LOGICAL BLOCK   9 | 8   LOGICAL DISPLACEMENT   0 |
+-------------------+------------------------+------------------------------+
   | 7                       | 8                      | 9
   |   +-----------+         v                        |
   |   | 128 MMU   |     +-------+                    |
   +-->| REGISTERS |---->| ADDER |                    |
       |           | 12  +-------+                    |
       +-----------+        | 12                      |
                            v                         v
             +-------------------------+-------------------------------+
             | 20   PHYSICAL BLOCK   9 | 8   PHYSICAL DISPLACEMENT   0 |
             +-------------------------+-------------------------------+
                            PHYSICAL ADDRESS (21 BITS)
```

The segment origin register (12 bits wide) contained the physical memory origin of the memory segment which was always on a 512 byte memory boundary. The logical address's segment value (upper 7 bits) was used as an index into the appropriate MMU origin register. The segment limit register (12 bits wide) contained both the segment size in terms of 512-byte pages (8 bits wide) and access control bits (4 bits wide) for the segment. The control bits determined what general area of memory was being accessed (main memory, I/O memory, or special I/O memory). The control bits also determined how the memory was used (read-only or as a stack).

If a memory-mapped I/O access is to a part of I/O memory that the emulator does not know about then the emulator should alert the user that an internal problem was found (the emulator debugger should come into play here). The emulator writer should then be notified and determine what is going on. In general, the emulator's memory access should always be to memory locations that the emulator is completely cognizant about. I recommend that there exist in the emulator a programming module called the Lisa Hardware Abstraction Layer. This module's purpose is to handle all hardware access to the emulator's host machine. For example, there should exist a routine here that reads the host machine's keyboard. When the emulator detects that the Lisa is trying to read the Lisa keyboard this module's keyboard routine would be called. Having this module is important since it segregates all host machine hardware access to a single area in the emulator which if the emulator is moved to another machine then only one area should need changing.

The Lisa's memory is accessed using the following physical memory map which shows what regions of physical memory hold what information (x in an address means don't-care, see the Lisa hardware manual for complete details):

Main RAM	0-1FFFFF (1MB)
Main RAM	0-2FFFFF (2MB)
Boot ROM	FExxxx (16KB)
I/O space	FCxxxx
Expansion slot 1	FC0000-FC3FFF
Expansion slot 2	FC4000-FC7FFF
Expansion slot 3	FC8000-FCBFFF
Floppy disk control	FCC001-FCCF77
Serial port control	FCD000-FCD3FF
Parallel port control	FCD800-FCDBFF
Keyboard/Mouse control	FCDC00-FCDFFF
CPU board control	FCE000-FCE01E
(unused)	FCE01F-FCE7FF
Video address latch	FCE8xx
Memory error address latch	FCF0xx
Status register	FCF8xx

Note: The Lisa also maintained a 128 byte chunk of memory whose contents were backed up by the Lisa's battery when the Lisa was powered-off. This memory was called Parameter Memory. I'm note sure where this memory resided in the above memory diagram but suspect it was in the I/O space.

Correct timing for the CPU and its instructions may be important for the emulator. For example, when the Lisa wants to read the Lisa's serial number (which is stored in the Lisa Video State ROM chip) access to the bits of this number occur at specific times only. It seems to me that accurate timing could be very difficult to achieve.

Note: Since the Video State ROM is not something that an emulator can access you may want to simulate this chip by having your code report whatever information the Lisa needs. For the serial number I recommend that you use a real serial number which I can provide from my Lisa.

All of the low-level controls for floppy and hard disks should be supported. I recommend supporting the Sony 400K floppy, the ProFile 5MB hard disk, and the Widget 10MB hard disk. I recommend against supporting the Twiggy 860K floppies that shipped with the Lisa 1 since I'm not sure if the last Lisa Office System (version 3.0 -- Lisa 7/7) supports Twiggies. This support in the emulator could be fairly tricky since the controls for disks seem complicated in the Lisa hardware manual. Also, support for the ProFile and Widget needs to follow a specific communications protocol (I have the docs detailing this protocol). Support for the floppy could be achieved using either of two strategies. You could emulate the really low-level hardware stuff such as the 6504 chip or you could just intercept calls to the floppy and use your host machine's OS to perform file-based disk I/O.

The Lisa can support two virtual terminals. In normal use (e.g. typical users using the Lisa Office System) there is only one screen, called the main screen. This is just a chunk of memory that is treated as a single bitmap by the Lisa's drawing code QuickDraw. But when the LisaBug debugger is installed there exists a second screen called the alternate screen. It is just another bitmap in memory. All LisaBug output is to this screen and it supports only text output even though it is at the lowest levels a bitmap.

HIGH-LEVEL KERNEL AREAS:

o Printing

The emulator utilities would cover the following:

o Emulator window handling (e.g. moving)
o Emulator disk image mounting, unmounting, and creating
o Emulator preferences
o Emulator access to Lisa disks
o Emulator debugger
o Emulator logging

Emulator access to Lisa disks may be extended from just providing the bare block-level accesses to file-level access. To achieve this the emulator would need to know the format of Lisa disks. This information, some of which I have, is fairly complicated since there were 3 different file formats supported by the Lisa over time. These file systems ranged from a flat-file system to a hierarchical system with various format changes in each main file system version. There was also a very old file system supported by the Lisa which ran under the Lisa Monitor OS. This was a UCSD-P System-based file system whose directory structure is specified in the Lisa Desktop Library interfaces. Though this FS is the oldest several of Apple's Lisa programs used it (e.g. LisaTest and MacWorks). I recommend that this feature allow the user to at least copy files from a mounted Lisa disk to the user's host machine FS.

Emulator logging seems like a very useful feature for the emulator writer. This feature would log all important events that occur within the emulator. This logging should occur to a file in the host machine's file system.

SECTION 7 -- LISA DISK IMAGES

The Lisa emulator should support the mounting, unmounting, and creation of disks just like the real Lisa does. See the USER INTERFACE section for a description of how the emulator user would deal with disks.

I recommend that no real Lisa disks be used with the emulator. Instead I recommend virtual disks be used. Specifically, I would use disk image files. For example, the Apple Macintosh computer can read and write Lisa disks at the block level using various Macintosh utility programs (e.g. Disk Copy from Apple Computer). These utilities can read a Lisa disk and can then create a file representing this disk on a Macintosh disk. The emulator should use these disks since they are platform neutral (e.g. these images are just files which non-Macintosh machines such as the PC can easily handle).

I recommend that the disk image files be created by a program that the emulator writer has complete programming control over. For example, I have a Macintosh disk utility program (named DTCMacDiskUtility) which can read and write its own image files and can read and write to Lisa floppy disks. The format of these files is documented and I have complete control over this program so that if bugs are detected or new features are needed I can resolve these.

The disk image files should support the Lisa 400K Sony floppies and the Apple ProFile 5MB or Apple Widget 10MB hard disks.

David Craig has disk image copies of all of the Lisa version 3 software which includes the Lisa Office System and the Lisa Workshop. David also has other Lisa disks with items like the Lisa Art Department (a clipart set) and the Lisa ToolKit development libraries and sources.

SECTION 8 -- PRINTING EMULATION

The emulator should also support printing.

Since the Lisa supported the Apple ImageWriter dot matrix printer and the Apple Daisy Wheel printer the emulator should support at least one of these. I recommend supporting the ImageWriter printer. To support these printers you will need to know the command sets for each printer (I have the reference manuals for these printers which contain this information). For the ImageWriter you would also need to know its character bitmaps (the reference manual contains these).

The emulator should print to whatever printer is attached to it using whatever printer driver is normally used. For example, the emulator could print to a laser printer yet the emulator would be simulating the ImageWriter. The output should look exactly like the original printer's output. For example, since the ImageWriter could print at a resolution of around 144 dpi the output from the emulator should look like 144 dpi printing (for laser printers that print at 300 or higher dpi today the emulator writer would need to scale the emulator printout to fit this dpi).

SECTION 9 -- DEBUGGER

The Lisa emulator should support a powerful built-in debugger. The purpose of this debugger is to give emulator writers a tool for analyzing and controlling emulator operations. Without this debugger creating an emulator would be a very difficult task. For example, this debugger would support access to any of the memory used by the emulator for emulation data structures such as the 1MB of RAM that the Lisa uses, the 68000 CPU registers, and the Lisa MMU registers.

I recommend that this debugger be a separate window that the user can display at any time. Display of this window could be through a special keypress (e.g. CONTROL-SHIFT-D). The user should interact with this debugger via a command-line interface since this type of control is fairly easy to implement, can easily be extended, and fits the psyche of many programmers. This window should mostly contain a simple text I/O area that covers the majority of the window. You could also have a small area containing all of the 68000 CPU registers and maybe some other stuff specific to the Lisa (e.g. the Lisa MMU domain value). This area could exist on the left side of the debugger window in the same way that Apple's MacsBug debugger window appears. I recommend against having a specific area for debugger commands, these should co-exist with the standard text I/O area.

Debugger activation should always display the emulator and debugger version numbers. This information could be useful when a person examines a debugger log and needs to know which version of the emulator or debugger made the debugger log.

Activation of the debugger should include a message from the emulator describing the reason for the activation. There would need to exist a message passing capability between the emulator and the debugger (in the Macintosh world there is such a capability called the DebugStr system call). After the activation message there should appear a dump of the Lisa's 68000 registers (see the D68 debugger command for a sample register dump). Activation messages should include the following:

USER ENTRY
This appears when the user activates the debugger.

MEMORY ACCESS VIOLATION
This appears when there is an access to memory which the emulator does not support.

68000 EXCEPTION
This appears when there is a 68000 CPU opcode error. For example, ADDRESS ERROR or ILLEGAL INSTRUCTION (see the 68000 manual or page 8-3 of the LisaBug documentation for a complete list). I'm not sure if these exceptions should be handled by the emulator since there may be another way to handle these (e.g., the emulator should pass these on to the Lisa exception handlers which will do whatever the Lisa normally does).

Activation should also display the message "Type EXIT to return to the Emulator" since this message could really help non-technical users who inadvertently enter the debugger.

The debugger should support the following types of commands:

- On-line help
- Version information
- Execution control
- 68000 CPU register and code access
- Memory access
- MMU register access
- Code tracing (with single stepping)
- Code timing
- Memory access trapping
- Logging and memory dumping
- Disk block dumping

The command set for the debugger could either be based on LisaBug's command set or it could be completely different. I recommend that the command set be different than LisaBug's commands so that the commands can be more understandable. There should also exist on-line command help so that the user need not have to read a manual to understand the commands. I also recommend that if an invalid command is entered by the user that the debugger tell the user the command is invalid and tell the user how to access the on-line command help.

Additional non-LisaBug commands could also be supported for special purposes. For example, writing all of the emulator memory data structures to a snapshot file which could be read into the emulator at a later time for analysis.

All debugger window output should be stored in a text file. The default text file is named "Lisa Emul Debug YYYYMMDD HHMMSS" where YYYYMMDD is the current date and HHMMSS is the current time in 24-hour format. This file is created when the debugger first starts and is closed when the emulator quits.

Debugger commands which can generate lots of screen output should be pausable and stoppable by the user. I recommend using the SPACE key as the pause and resume key and the ESCAPE key or a control (or Apple) key press in conjunction with a standard key such as Q.

Each debugger command name has a full name and an abbreviated name. Only the abbreviated names are understood by the debugger. Command character case should be immaterial. Command names may be followed by parameters (character case here is also immaterial). Optional parameters are enclosed in {}. All command value parameters are in hexadecimal (e.g. 10 means decimal 16, A means decimal 10).

I recommend the following Lisa Emulator Debugger commands:

ON-LINE HELP COMMANDS

o ? / HELP {command}

Displays detailed on-line help information for the specified command abbreviation. If command is not present then lists all of the commands showing a single line of help for each command.

Example: ? V

Display version information for both the Lisa emulator and the debugger.

VERSION INFORMATION COMMANDS

o V / VERSION

Displays version information for both the Lisa emulator and the debugger. Version information has the format A.BC.DE where A is the main version, BC is the revision level (changes to functionally have been made), and DE is the bug fix level. There should also be a date associated with each version number (dates should have a universal format such as YYYY-MM-DD, e.g. 1969-07-20 represents 20 July 1969).

Example: V

```
Lisa Emulator Version 0.00.00 [1969-07-20]
Debugger Version 0.00.00 [1969-07-18]
```

EXECUTION CONTROL COMMANDS

o EXIT (or BYE)

Example: EXIT

Returns to the emulator.

o G / GO {address}

Starts execution of 68000 code at the address. If address is not present then execute at the current 68000 program counter.

68000 CPU REGISTER AND CODE ACCESS COMMANDS

o   D68 / DUMP 68000 REGISTERS

Displays the 68000 CPU registers.

Example: D68

```
PC=xxxxxxxx  SR=xxxxxxxx  US=xxxxxxxx  SS=xxxxxxxx  DO=x
D0=xxxxxxxx  D1=xxxxxxxx  D2=xxxxxxxx  D3=xxxxxxxx
D4=xxxxxxxx  D5=xxxxxxxx  D6=xxxxxxxx  D7=xxxxxxxx
A0=xxxxxxxx  A1=xxxxxxxx  A2=xxxxxxxx  A3=xxxxxxxx
A4=xxxxxxxx  A5=xxxxxxxx  A6=xxxxxxxx  A7=xxxxxxxx
```

Note: SR is the Status Register, US is the User Stack pointer, SS is the Supervisor Stack pointer, and DO is the Lisa MMU domain (I'm including this here since it is useful to know about).

o   S68 / SET 68000 REGISTER register value

Sets the 68000 CPU register to the specified value. Address registers are specified as A0 to A7, data registers as D0 to D7, program counter as PC, status register as SR, and the supervisor stack pointer as SS (A7 is the user stack pointer which can also be specified as US).

Example: S68 A0 12345678

Sets 68000 register A0 to 12345678.

o   D / DISASSEMBLE {start-addr} {end-addr}

Disassembles 68000 code starting from address start-addr to end-addr. If only 1 address is present then disassemble 20 instructions from start-addr. If no addresses are present then disassemble 20 instructions starting at the instruction just after the last disassembled instruction. The disassembled output contains the address, hex opcodes, ASCII for the opcodes, and disassembled instructions. Invalid opcodes display "???". Note that "$" in front of a number means hexadecimal (this is the standard way Motorola assemblers/debuggers show these numbers so I'm doing the same here).

Example: D 100 10D

```
00000100: 4A62 EFF2           'Rt..'    TST.W    $EFF2(A7)
00000104: 4E56 FFF2           'W6..'    LINK     A6,#$FFF2
00000108: 3D7C 0009 FFFE      'C$....'  MOVE.W   #$0009,$FFFE(A6)
```

MEMORY ACCESS COMMANDS

o   DM / DUMP MEMORY {start-addr} {end-addr}

Displays memory in a standard hex/ascii dump format beginning at start-addr and ending at end-addr. If end-addr is not entered then displays 64 bytes. If start-addr is not specified then display the next 64 bytes. Each outputted line always contains 16 bytes displayed first as hex values then followed by 16 characters representing those 16 bytes' ASCII values. ASCII values not in the range $21-$7E are printed as ".". The memory can be any area of the Lisa's memory including main memory, ROM memory, or I/O memory.

Example: DM 0 10

```
000000: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F ....XYZ..5rfU..@
```

o   SM / SET MEMORY start-addr values

Sets main memory starting at start-addr to the values contained in values.

Example: SM 0 11223344

Sets memory starting at address 0 so that address 0 contains 11, address 1 contains 22, address 2 contains 33, and address 3 contains 44.

o   FM / FIND MEMORY start-addr end-addr pattern

Finds patterns in memory. The pattern can be any number of bytes (but it must contain an even number of hexadecimal digits).

Example: FM 0 1FFFFF A900

Searches memory in the range 0-1FFFFF for the word pattern A900.

o   CM / COMPARE MEMORY start-addr end-addr dest-addr

Compares the bytes in the range start-addr to end-addr against the bytes starting at dest-addr. Any mismatches produce the message "Mismatch starting at address x". A complete match produces the message "Equal."

Example: CM 0 1FF 500

```
Mismatch starting at address 123          (for a mismatch)
Equal                                     (for a match)
```

MMU REGISTER ACCESS COMMANDS

o   DMMUD / DUMP MMU DOMAIN

Displays the current MMU domain value (range: 0-3).

Example: DMMUD

```
DOMAIN 0
```

o   DMMU / DUMP MMU REGISTERS domain {start-reg} {end-reg}

Displays the Lisa MMU registers for the domain (0-3) starting at start-reg and ending at end-reg with register range being 0-7F. If one register exists then only that register's fields are listed. If no registers exist then all registers are displayed. If domain is "ALL" then all 4 domain MMU register sets are displayed.

Example: DMMU 0 7F

```
DOMAIN 0:  MMU REGISTER 7F -- Origin xxx   Limit xx   Control x   info
```

Note: Info describes the access controls for the segment in language such as "Read-Only", "Read-Write", or "Stack".

o SMMU / SET MMU REGISTERS domain register origin limit control

Sets the MMU register values for the specified domain (0-3). Register must be in the range 0-7F. Origin is a 3 nibble value (0-FFF), limit is a 2 nibble value (00-FF), and control is a single nibble value (0-F). This command can be a very dangerous command so be super careful here.

Example: SMMU 2 7F 12A 5F C

Sets register 7F in domain 2 so that the origin is 12A, the limit is 5F, and the control is C.

o LP / LOGICAL TO PHYSICAL address

Displays the physical address corresponding to the logical address. Also shows the 3 components of a logical address from the MMU's perspective. Each component value is listed in hexadecimal (e.g. $09), decimal (e.g. 9), and binary (e.g. %0001001). This will be very useful for testing the emulator's MMU handling.

Example: LP 123456

```
LOGICAL ADDRESS = 123456       PHYSICAL ADDRESS = FC379A
LOGICAL ADDRESS COMPONENTS:
  SEGMENT NUMBER       = $09  /   9 / %0001001
  LOGICAL BLOCK NUMBER = $1A  /  26 / %00011010
  LOGICAL DISPLACEMENT = $056 /  86 / %001010110
```

Note: $123456 has the binary value %000100100011010001010110.

CODE TRACING COMMANDS

o T / TRACE

Executes the next 68000 instruction and displays the 68000 registers (see command D68 for an example of the register dump). To trace more instructions just press the RETURN or ENTER keys.

Example: T

o BP / SET BREAK POINT address

Sets a break point at the address. Up to 100 break points are supported and are stored in a table. Duplicate break point addresses are detected and not added to the table. When the emulator attempts to execute the instruction at an address that resides in the break point table the debugger stops program execution, displays the message "BREAK POINT ENCOUNTERED", displays a dump of the 68000 registers, and waits for the user to enter a command.

Example: BP 123456

o   DBP / DUMP BREAK POINT TABLE

Dumps all of the defined addresses in the break point address table.

Example: DBP

```
 # ADDRESS
-- --------
 0 00123456
 1 00005000
```

o   CBP / CLEAR BREAK POINT {break-point#}

Clears a break point. If break-point# is not present then clear all the break points (it asks you first if this is OK). Use the DBP command to get the break-point#.

```
Example:    CBP 4                  (clears break point # 4)
            CBP                          (clears all the break points)
```

CODE TIMING COMMANDS

o   TB / SET TIMING BUCKET start-addr end-addr

Sets a timing bucket between addresses start-addr and end-addr. When the emulator encounters address start-addr it starts a 1/100 second timer until end-addr is reached. Up to 100 timing buckets can be defined.

Example: TB 100 200   (sets a timing bucket for addresses 100-200)

o   DTB / DUMP TIMING BUCKETS

Displays all of the defined timing buckets showing their starting address and their ending address.

Example: DTB

```
#   START     END
--  --------  --------
 0  00123400  001234FF
```

o   CTB / CLEAR TIMING BUCKET {bucket#}

Clears the timing bucket specified by bucket bucket#. If bucket# is not present then clears all the timing buckets (it asks you first if this is OK). Use the DTB command to get bucket#.

Example: CTB 2

o   RTB / RESET TIMING BUCKET {bucket#-start} {bucket#-end}

Resets the count and the elapsed time for a range of timing buckets. If only 1 timing bucket is specified then resets only that bucket's information. If no buckets are specified it resets all the buckets (it asks you first if this is OK). Use the DTB command to get bucket#.

```
Example:    RTB 6             (resets bucket # 6)
            RTB 6 10                (resets buckets # 6 to 10)
            RTB                     (resets all the buckets)
```

o DTBR / DUMP TIMING BUCKET REPORT

Displays a report detailing the time and number of times a timing bucket was entered. Time is in 1/100 second increments.

```
Example:    DTBR

#   START     END       COUNT     TIME
--  --------  --------  --------  --------
 0  00123400  001234FF        21       125
```

MEMORY ACCESS TRAPPING COMMANDS

o TM / TRAP MEMORY access-mode start-addr {end-addr}

Traps a memory access to any address within the address range. If end-addr is not present then traps only accesses to the byte at addr-start. Access-mode is either "R" for read-access, "W" for write-access, or "B" for both read- or write-access. When the emulator encounters an access to this address range it activates the debugger, displays the message "MEMORY TRAP - READ" or "MEMORY TRAP - WRITE" followed by the memory address which was trapped. Up to 100 memory traps may be defined. This is a very powerful command which emulator designers will use a lot.

```
Example:    TM R FC0000 FC00FF
```

Traps any memory reads to addresses in the range FC0000-FC00FF.

o TMV / TRAP MEMORY VALUE
access-mode start-addr end-addr value

Traps accesses to the address range but only for the specified value parameter. See the TM command for a general discussion of memory trapping. The number of hex digits in value determines whether the access is byte, word, or long based.

```
Example:    TMV W 100 1FF AAAA
```

Traps any write of the word value $AAAA to address range 100-1FF.

o TMP / TRAP MEMORY PATTERN
access-mode start-addr end-addr pattern

Traps accesses to the address range but only for the specific pattern parameter. See the TMV command for general information. The pattern is bit-based and specifies the bit pattern that triggers the trap. The pattern must be 8 bits long for byte trapping, or 16 bits long for word trapping, or 32 bits long for long trapping. A don't-care bit value of "x" may be specified.

```
Example:    TMP W 100 1FF 00001111   (traps if byte $0F written)
            TMP W 100 1FF 0xxxxxx   (traps if byte with high bit = 0)
```

o   DTM / DUMP TRAP MEMORY TABLE

Displays all of the memory defined traps. See the TM, TMV, and TMP commands.

Example: DTM

```
 #  START     END       MODE  VALUE     PATTERN
--  --------  --------  ----  --------  --------------------------------
 0  00FC0000  00FC00FF     R      FFFF
 1  00000100  000001FF     W                                      0xxxxxxx
```

o   CTM / CLEAR TRAP MEMORY {trap#}

Clears the memory trap specified by trap#. If trap# is not present then clears all the memory traps (it asks you first if this is OK). Use the DTM command to determine the trap#.

```
Example:    CTM 23                        (clears memory trap # 23)
            CTM                           (clears all the memory traps)
```

<u>LOGGING AND MEMORY DUMPING COMMANDS</u>

o   LOG / LOG {file-name}

Starts logging all debugger information to the text file named file-name. The current log file is first closed. If file-name argument is not entered then closes the current log file. If the file name contains spaces then delimit the name in either double (") or single (') quotes.

Example: LOG foobar.text

Creates a new log file named "foobar.text".

o   DEM / DUMP EMULATOR MEMORY file-name

Dumps all the emulator's Lisa memory data structures to the binary file named file-name. This is a snapshot facility that can be very useful for debugging an emulator that has gone nuts. This file should contain the 68000 registers, the Lisa's main memory, the Lisa's MMU registers, the I/O registers and any other memory that this emulator maintains for the Lisa environment. If the file name contains spaces then delimit the name with either double (") or single (') quotes (e.g. "My Tag File" or 'My Tag File').

Note: I recommend that this binary file contain a short header that indicates this file is a Lisa Emulator memory snapshot. Following this header should appear tagged data objects. These objects have a short header which specifies the type of data object it is (e.g. 68000 registers) and the size of the object (e.g. 100 bytes). Having a tagged structure simplifies the creation and modification of these files in case changes need to be made the changes (e.g. a new object) should be upwardly compatible. Tagged object data files have been used successfully in the past, e.g. the TIFF (Tagged Image File Format) which I believe was developed by Microsoft and Aldus to store image data in a machine-neutral fashion.

I recommend the following detailed tagged object file format:

```
+------------------------+
|      HEADER OBJECT     |
+------------------------+
|      OBJECT 1          |
+------------------------+
|      ...               |
+------------------------+
|      OBJECT N          |
+------------------------+
|      FOOTER OBJECT     |
+------------------------+
```

The **HEADER OBJECT** must be the first object in the file. The **FOOTER OBJECT** must be the last object in the file. The other objects may be in any order. I recommend the following tags for this file ("*" should be used in the tag name so that the tag is always 16 bytes long, I also recommend that each tag name contain only human-readable characters):

Object	Tag	Contents
Header	LisaEmulSnapshot	Header object (always first)
Footer	LisaEmulFinis***	Footer object (always last)
68000	MC68000*********	68000 registers
MMU	LisaMMU*********	Lisa MMU registers
Main Mem	LisaMainMemory1M	Lisa main memory -- 1 MB
Main Mem	LisaMainMemory2M	Lisa main memory -- 2 MB
Screen Main	LisaScreenMain**	Address of Lisa's main screen
Screen Alt	LisaScreenAlt***	Address of Lisa's alternate screen
ROM Mem	LisaBootROM*****	Lisa boot ROM image (16KB)
I/O Space	LisaIOMemory****	Lisa I/O memory

Note: A file should normally contain only one Main Mem object corresponding to the size of main memory. The Screen Main and Screen Alt tags exist so that it is easy to access the main and alternate (if present) screen areas in the Lisa Mem object for debugging or whatever purpose.

A tagged object should have the following general format:

```
+-----------------------------------------+
|  OBJECT TAG                 (16 BYTES)  |
+-----------------------------------------+
|  OBJECT DATA BYTE LENGTH L  ( 4 BYTES)  |
+-----------------------------------------+
|  OBJECT DATA CHECKSUM       ( 4 BYTES)  |
+-----------------------------------------+
|  FLAGS                  [1] ( 4 BYTES)  |
+-----------------------------------------+
|  DATA ENCRYPTION METHOD [2] ( 2 BYTES)  |
+-----------------------------------------+
|  RESERVED               [3] (12 BYTES)  |
+-----------------------------------------+
|  OBJECT HEADER CHECKSUM [4] ( 4 BYTES)  |
+-----------------------------------------+
|  OBJECT DATA                ( L BYTES)  |
+-----------------------------------------+
```

Note [1]: The FLAGS field can hold up to 32 flags. None are currently defined so this field should contain just the value 0. Future flag use could encompass flags such as a read-only flag or a encryption flag.

Note [2]: The RESERVED field should contain 0s.

Note [3]: The DATA ENCRYPTION METHOD determines if the object data is encrypted. If this value contains 0 then no encryption is used (i.e. plain-text data). Other values could be defined for different types of encryption.

Note [4]: The OBJECT HEADER CHECKSUM should be a checksum of all of the object fields above this field. I recommend that the checksum be simple to compute yet contains radically different results for similar data. I have had lots of success with the following simple checksum algorithm which checksums a set of 4 byte long values (to checksum a set of bytes you would need to change this algorithm from long accesses to byte accesses):

```
1.  Init checksum to $6E8E8A00 (%01101110100011101000101000000000)
2.  For each long value do
      a)    Exclusive-OR the checksum and the value
      b)    Rotate left 31 bits the value from step a)
      c)    The value from step b) is the current checksum
3.  Return the final checksum value
```

For example, this algorithm for the single long 0 produces a checksum value of $37474500. For the long value 1 the checksum is $B7474500.

An implementation in Apple MPW Pascal of this algorithm is:

```
{ return a checksum for a list of longints based on an inputted checksum:

  _longs    = pointer to first long
  _count    = number of longs
  _checksum = input and output checksum value }

  procedure CalcChecksum (_longs: Ptr; _count: integer; var _checksum: longint);

  type tLongPtr = ^longint;

  var i: integer;
      l: longint;

  begin
    i := 0;

    while _count > 0 do begin
      l := tLongPtr(ord4(_longs)+(i*4))^;
      _checksum := BROTL( BXOR(_checksum,l) , 31 );
      i := i + 1;
      _count := _count - 1;
    end;
  end;
```

The HEADER OBJECT should contain the following data fields:

BYTE ORDER [1]	2 BYTES	"LL" for Little-Endian, "BB" for Big-Endian
CREATION DATE	10 BYTES	"YYYY-MM-DD"
CREATION TIME	8 BYTES	"HH:MM:SS" in 24-hour time
CREATION PROGRAM	64 BYTES	Name of program that created this file
PROGRAM VERSION	64 BYTES	Version number and compile date of program
ACCESS PASSWORD [2]	16 BYTES	Access password
COMMENTS [3]	256 BYTES	Any comments the program wants here
RESERVED	92 BYTES	Reserved for future use -- fill with 0s

Note [1]: The BYTE ORDER field is very important since it specifies how multi-byte integral values are physically stored in the file. If your emulator's host machine is running a Big-Endian CPU such as a 68000 family chip then most likely BYTE ORDER will contain "BB". If your host CPU is an Intel chip then most likely the BYTE ORDER will be ""LL". But this ordering as determined by the BYTE ORDER field _only_ applies to the tagged object file and the computer that creates this file can use whatever order it wants as long as it defines BYTE ORDER appropriately. Also note that BYTE ORDER applies to _all_ of the objects in the file and not just the header object. Also, the byte order does not apply to the contents of the data field in an

object, this field's contents are explicity defined by the object containing this data.

Note [2]: The ACCESS PASSWORD field contains a password string (with up to 16 characters) that the creator has assigned to this file for access control. The password itself should be encrypted in some fashion so that viewing a hex dump of this file won't show the password as plain-text. I recommend a simple password encryption scheme such as exclusive-ORing each byte with a string of known bytes. If this field contains all 0s then no password exists. I also recommend that this password not be case-sensitive (why? I hate case-sensitive passwords). This is not a Pascal string.

Note [3]: The COMMENTS field should contain a Pascal-style string. This means the first byte of the string contains the string's length in characters. The rest of the string contains the actual string characters. A 256 byte Pascal string can therefore hold 255 characters maximum which should provide plenty of space for a comment.

Example: DEM foobar.memory

Dumps emulator memory to a file named "foobar.memory".

o LEM / LOAD EMULATOR MEMORY file-name

Loads a saved emulator dump file. This replaces all of the active emulator data structures, displays the 68000 registers, and then waits for you to type a debugger command. You can then examine these structures with debugger commands and then continue execution.

Example: LEM foobar.memory

o PICT / WRITE SCREEN PICTURE

Creates a picture file of the Lisa emulator screen to files named "Lisa Emul Picture NNNN.suffix" where NNNN is a number and suffix is a suffix (e.g. "Lisa Emul Picture 0001.pict"). The picture file could be a standard file for the host machine. For example, if the host is a Macintosh then produce standard PICT files (use the suffix ".pict"). If the host is a Windows machine then produce Bitmap files (use suffix ".bmp"). This feature could be useful for easily documenting Lisa emulator screens. I also recommend that the Lisa emulator support a keypress that calls this command. Or, if you want this file to be machine-neutral then store the screen in its own format, such as a simple bitmap. If you go this route I recommend that you use a tagged object file for the image file (this file could contain just 3 tags; a header tag, a footer tag, and a tag for the image data).

Example: PICT

o PICTP / WRITE SCREEN PICTURE PERIODICALLY period

Same as the PICT command but does not immediately produce a picture file. Instead, this command tells the debugger (or the emulator) to periodically produce a picture file based on the period parameter. Period represents the number of seconds that each picture should be taken.

Example: PICTP 60 (periodically produce a picture every 60 seconds)

DISK BLOCK DUMPING

o DMD / DUMP MOUNTED DISKS

Displays a list of all the mounted emulator disk images. Each mounted disk is assigned a unique disk number which is used by the other disk block commands. The display shows the disk number and the disk size in 512 byte blocks.

Example: DMD

```
#  SIZE
-- --------
 0      800
 1      800
```

o DDB / DUMP DISK BLOCK disk# {start-block#} {end-block#}

Displays the contents of the specified 512 byte block range from the specified disk in a standard hex/ASCII format. If no blocks present then dumps all of the disk blocks. Also includes the block's 24 byte TAG (or 12 bytes if the full 24 are not available - the Macintosh can read Lisa disks but can only read 12 bytes of the TAG). Use the DMD command to get the disk#.

```
Example:    DDB 1  200 300        (dumps blocks 200-300 from disk 1)
```

SECTION 10 -- IMPLEMENTATION LANGUAGE

The implementation language should be a high-level language so that the emulator is as portable as possible. Potential language choices are C (or C++) or Pascal or any other high-level language that can handle the emulator's software needs.

I recommend against using assembly language for the following reasons:

o Assembly language is inherentaly non-portable between machines.

o Assembly language is much harder to maintain.

o Assembly language will most likely not provide any significant speed increases since today's machines with 300 MHz CPUs will be much faster than any emulator really needs no matter the language.

SECTION 11 -- IMPLEMENTATION MACHINE

Any computer that supports the following hardware should be able to support an emulation of the Lisa:

o Bit-mapped screen. Resolution must be at least 720h x 364v pixels. The machine can support color pixels but the Lisa supported only B/W display output.

o Mouse. The Lisa supported only a single button mouse. Multi-button mice could be used by the Lisa emulator (you could use the emulation mouse so only 1 button acted as the Lisa's single-button or all buttons could work like the Lisa's mouse button).

o   At least 1 MB of memory (2MB was also supported). Since the Lisa supported up to 1MB of RAM the emulator for simplicity sake should also support this. Additional memory would be needed by the emulator itself. If 1MB of RAM for the Lisa itself is not doable then you could always use a virtual-memory scheme (I bet that a VM scheme would not slow down an emulator given the speed of today's hard disks or even floppies).

o   Keyboard with special keys. The Lisa keyboard contained an APPLE key and an OPTION key. The APPLE key was used for keyboard-based commands. The OPTION key was used to generate foreign or special characters. Today's Macintosh keyboards with the Apple (or Command) key and the Option key would work. Today's PC keyboards with the Control key and the Alt key would also work.

o   Fairly fast CPU and hard disk. The Lisa's CPU ran at 5 MHz and the peripherals were fairly slow. Therefore if you want an emulator to be at least as fast as the Lisa you should use a machine that is much faster than the Lisa. Any of today's machines such as Pentium-based Windows machines or 68020/30/40/PowerPC Macintosh machines should work well. You may actually have to slow down the emulator if it works too fast.

SECTION 12 -- FUN STUFF

To make this emulator have some spirit I recommend that it also contain historical and technical information related to the Lisa. For example, there could be included in this emulator a history of the Lisa such as my Lisa Legacy paper. There could also be scanned Lisa documents such as Apple's Lisa Architecture paper and/or the Lisa designer interview from BYTE magazine. This information would provide the curious reader with some background information about the Lisa and its place in computer history. Displaying a picture of the Lisa during emulator startup would also be a nice touch (the Lisa image that originated with LisaDraw and which heads my Lisa paper could be a good candidate).

SECTION 13 -- REFERENCES

The following references should prove valuable in understanding the Lisa hardware and software. The ability to read Pascal programming code is also needed since the Lisa's software was for the most part written in Lisa Pascal. You must also be able to at least read 68000 assembly language programming.

The Architecture of the Lisa Personal Computer
Apple Computer, 1984, 12 pages

This is a great technical summary of the Lisa's hardware and software architecture which was written by one of the Lisa's designers.

Lisa Product Data Sheets
Apple Computer, 1983-1984

These product sheets provide lots of concise information about the Lisa's hardware and software. Each sheet is around 4 pages long. I have the sheets for the Lisa 1 hardware, Lisa 2 hardware, Lisa user tools (e.g. LisaDraw), Lisa programming tools (e.g. LisaPascal), and peripherals (e.g. ImageWriter printer).

Lisa Owner's Guide
Apple Computer, 1983-1984

The Lisa 1 Owner's Guide and the Lisa 2 Owner's Guide are very useful since they provide detailed information about how the Lisa physically looked and how you performed basic tasks with it. If you're not familiar with the Lisa or don't have these manuals I highly recommend them. Also highly recommended are the Lisa tool manuals (e.g. LisaWrite) since these explain the details behind these tools.

The Mac Bathroom Reader
Owen Linzmayer, 1994

For an extremely accurate and readable description of the Lisa's history see this book's chapter "Lisa: From Xerox with Love." If you read only a single history about the Lisa (or Apple Computer in general) make it this book. Owen is currently updating this book. Contact Owen for book availability info:

e-mail:	owenink@ix.netcom.com
WWW:	http://www.netcom.com/~owenink/reviews.html/

Lisa Hardware Manual
Apple Computer, 1981 and 1983

This manual is the definitive description of the Lisa 1 computer's hardware. The 1983 revision supersedes the 1981 manual and provides lots of information about the Lisa hardware interfaces which an emulator writer will definitely need. The 1981 manual does describe a few items in a little more detail than this manual (e.g. the MMU info in the 1981 manual seems a little more complete). The 1981 manual is 82 pages long, the 1983 manual is 333 pages long.

Lisa Boot ROM Listing
Apple Computer

This is the listing for the Lisa's boot ROM (version 2.48 "H") which was written in 68000 assembly language. This is a must-have document for emulator writers since it shows how the boot ROM interfaces with the Lisa hardware to both test this hardware and to boot an OS from a disk device. If the hardware manual is unclear about a certain aspect of the hardware this listing will most likely have the answer. I have this listing as a disk file.

As an example of the contents of the Boot ROM listing here's the mouse handling code. Note that the Boot ROM is in general very well commented.

```
2F2A|                    ;------------------------------------------------------------------------
2F2A|                    ;
2F2A|                    ;   Hardware Interface for the Mouse
2F2A|                    ;
2F2A|                    ;   Written by Rick Meyers
2F2A|                    ;   (c) Apple Computer Incorporated, 1983
2F2A|                    ;
2F2A|                    ;   The routines below provide an assembly language interface to the mouse.
2F2A|                    ;   Input parameters are passed in registers, output parameters are returned
2F2A|                    ;   in registers.  Unless otherwise noted, all registers are preserved.
2F2A|                    ;
2F2A|                    ;   The Mouse
2F2A|                    ;
2F2A|                    ;   The mouse is a pointing device used to indicate screen locations.  Mouse
2F2A|                    ;   coordinates are located between pixels on the screen.  Therefore, the
2F2A|                    ;   X-coordinate can range from 0 to 720, and the Y-coordinate from 0 to 364.
2F2A|                    ;   The initial mouse location is 0,0.
```

```
2F2A|                       ;
2F2A|                       ;   Mouse Scaling
2F2A|                       ;
2F2A|                       ;   The relationship between physical mouse movements and logical mouse
2F2A|                       ;   movements is not necessary a fixed linear mapping.  Three alternatives
2F2A|                       ;   are available: 1) unscaled, 2) scaled for fine movement and 3) scaled
2F2A|                       ;   for coarse movement.
2F2A|                       ;
2F2A|                       ;   When mouse movement is unscaled, a horizontal mouse movement of x units
2F2A|                       ;   yields a change in the mouse X-coordinate of x pixels.  Similiarly, a
2F2A|                       ;   vertical movement of y units yields a change is the mouse Y-coordinate
2F2A|                       ;   of y pixels.  These rules apply independent of the speed of the mouse
2F2A|                       ;   movement.
2F2A|                       ;
2F2A|                       ;   When mouse movement is scaled, horizontal movements are magnified by 3/2
2F2A|                       ;   relative to vertical movements.  This is intended to compensate for the
2F2A|                       ;   2/3 aspect ratio of pixels on the screen.  When scaling is in effect, a
2F2A|                       ;   distinction is made between fine (small) movements and coarse (large)
2F2A|                       ;   movements.  Fine movements are slightly reduced, while coarse movements
2F2A|                       ;   are magnified. For scaled fine movements, a horizontal mouse movement of
2F2A|                       ;   x units yields a change in the X-coordinate of x pixels, but a vertical
2F2A|                       ;   movement of y units yields a change of (2/3)*y pixels.  For scaled coarse
2F2A|                       ;   movements, a horizontal movement a x units yields a change of (3/2)*x
2F2A|                       ;   pixels, while a vertical movements of y units yields a change of y pixels.
2F2A|                       ;
2F2A|                       ;   The distinction between fine movements and coarse movements is determined
2F2A|                       ;   by the sum of the x and y movements each time the mouse location is
2F2A|                       ;   updated.  If this sum is at or below the 'threshold', the movement is
2F2A|                       ;   considered to be a fine movement.  Values of the threshold range from 0
2F2A|                       ;   (which yields all coarse movements) to 256 (which yields all fine
2F2A|                       ;   movements).  Given the default mouse updating frequency, a threshold of
2F2A|                       ;   about 8 (threshold's initial setting) gives a comfortable transition between
2F2A|                       ;   fine and coarse movements.
2F2A|                       ;---------------------------------------------------------------------------------
2F2A|
2F2A|
2F2A|                       ;---------------------------------------------------------------------------
2F2A|                       ;
2F2A|                       ;   Mouse Movement
2F2A|                       ;
2F2A|                       ;   This routine is called by the GETINPUT routine when the COPS has
2F2A|                       ;   reported mouse movement.  All registers are preserved.
2F2A|                       ;
2F2A|                       ;
2F2A|                       ;   Register Assignments:
2F2A|                       ;
2F2A|                       ;       D0  --  Mouse X-Coordinate (integer)
2F2A|                       ;       D1  --  Mouse Y-Corrdinate (integer)
2F2A|                       ;       D2  --  Mouse Dx (integer)
2F2A|                       ;       D3  --  Mouse Dy (integer)
2F2A|                       ;
2F2A| 48E7 7C00             MouseMovement   MOVEM.L D1-D5,-(SP)         ; save registers
2F2E| 3038 0486                             MOVE.W  MousX,D0            ; mouse X-coordinate
2F32| 3238 0488                             MOVE.W  MousY,D1            ; mouse Y-coordinate
2F36| 1438 048A                             MOVE.B  MousDx,D2           ; mouse Dx (byte)
2F3A| 4882                                  EXT.W   D2                  ; mouse Dx (integer)
2F3C| 1638 048B                             MOVE.B  MousDy,D3           ; mouse Dy (byte)
2F40| 4883                                  EXT.W   D3                  ; mouse Dy (integer)
2F42|
2F42| 3802                  Scale           MOVE.W  D2,D4               ; mouse Dx
2F44| 6C02                                  BGE.S   @1                  ; branch if >= 0
2F46| 4444                                  NEG.W   D4                  ; ABS(mouse Dx)
2F48|
2F48| 3A03                  @1              MOVE.W  D3,D5               ; mouse Dy
2F4A| 6C02                                  BGE.S   @2                  ; branch if >= 0
2F4C| 4445                                  NEG.W   D5                  ; ABS(mouse Dy)
2F4E|
2F4E| D845                  @2              ADD.W   D5,D4               ; ABS(Dx) + ABS(Dy)
2F50| 9878 048E                             SUB.W   MousThresh,D4       ;  - MouseThreshold
2F54| 6E16                                  BGT.S   Coarse              ; branch if coarse movement
2F56|
2F56| D042                  Fine            ADD.W   D2,D0               ; new X-coordinate (scale 1)
2F58| 3403                                  MOVE.W  D3,D2               ; save Dy
2F5A| D643                                  ADD.W   D3,D3               ; Dy*2
2F5C| D643                                  ADD.W   D3,D3               ; Dy*4
2F5E| D642                                  ADD.W   D2,D3               ; Dy*5
2F60| 5443                                  ADDQ    #2,D3               ; (Dy*5)+2
```

```
2F62| 6D02                            BLT.S   @3                 ; branch if negative
2F64| 5643                            ADDQ    #3,D3              ; (Dy*5)+5
2F66| E643              @3            ASR.W   #3,D3              ; Dy*(5/8) with rounding
2F68| D243                            ADD.W   D3,D1              ; new Y-coordinate (scale 5/8)
2F6A| 6010                            BRA.S   Bounds             ; continue
2F6C|
2F6C| D243              Coarse        ADD.W   D3,D1              ; new Y-coordinate (scale 1)
2F6E| 3602                            MOVE.W  D2,D3              ; save Dx
2F70| D443                            ADD.W   D3,D2              ; Dx*2
2F72| D443                            ADD.W   D3,D2              ; Dx*3
2F74| 6D02                            BLT.S   @4                 ; branch if negative
2F76| 5242                            ADDQ.W  #1,D2              ; (Dx*3)+1
2F78| E242              @4            ASR.W   #1,D2              ; Dx*(3/2) with rounding
2F7A| D042                            ADD.W   D2,D0              ; new X-coordinate (scale 3/2)
2F7C|
2F7C| 4A40              Bounds        TST.W   D0                 ; new X-coordinate >= 0
2F7E| 6C02                            BGE.S   @5                 ; branch if >= 0
2F80| 4240                            MOVE.W  #0,D0              ; minimum X of 0
2F82|
2F82| 0C40 02D0         @5            CMP.W   #MaxX,D0           ; new X-coordinate <= 720
2F86| 6F04                            BLE.S   @6                 ; branch if <= 720
2F88| 303C 02D0                       MOVE.W  #MaxX,D0           ; maximum X of 720
2F8C|
2F8C| 4A41              @6            TST.W   D1                 ; new Y-coordinate >= 0
2F8E| 6C02                            BGE.S   @7                 ; branch if >= 0
2F90| 4241                            MOVE.W  #0,D1              ; minimum Y of 0
2F92|
2F92| 0C41 016C         @7            CMP.W   #MaxY,D1           ; new Y-coordinate <= 364
2F96| 6F04                            BLE.S   @8                 ; branch if <= 364
2F98| 323C 016C                       MOVE.W  #MaxY,D1           ; maximum Y of 364
2F9C|
2F9C| 31C0 0486         @8            MOVE.W  D0,MousX           ; update Mouse X-coordinate
2FA0| 31C1 0488                       MOVE.W  D1,MousY           ; update Mouse Y-coordinate
2FA4| 31C0 0496                       MOVE.W  D0,CrsrX           ; also update cursor coordinates
2FA8| 31C1 0498                       MOVE.W  D1,CrsrY
2FAC| 4CDF 003E                       MOVEM.L (SP)+,D1-D5        ; restore registers
2FB0| 4E75                            RTS                        ; return
2FB2|
2FB2|                   ;---------------------------------------------------------------------------
2FB2|                   ;
2FB2|                   ;  Routine to initialize mouse handling
2FB2|                   ;
2FB2|                   ;---------------------------------------------------------------------------
2FB2|
2FB2|                   MousInit
2FB2| 31FC 0168 0486                  MOVE.W  #360,MousX         ; set inital mouse location
2FB8| 31FC 00B6 0488                  MOVE.W  #182,MousY         ;  to center of screen
2FBE| 31FC 0008 048E                  MOVE.W  #8,MousThresh      ; set scaling threshold
2FC4| 707C                            MOVEQ   #$7C,D0            ; and enable mouse data
2FC6| 6100 D98E                       BSR     COPSCMD
2FCA| 4E75                            RTS
2FCC|
2FCC|                   ;---------------------------------------------------------------------------
2FCC|                   ;
2FCC|                   ;   Hardware Interface for the Cursor
2FCC|                   ;
2FCC|                   ;   Written by Rick Meyers
2FCC|                   ;   (c) Apple Computer Incorporated, 1983
2FCC|                   ;
2FCC|                   ;
2FCC|                   ;   The routines below provide an assembly language interface to the cursor.
2FCC|                   ;   Input parameters are passed in registers, output parameters are returned
2FCC|                   ;   in registers.  Unless otherwise noted, all registers are preserved.
2FCC|                   ;
2FCC|                   ;   The Cursor
2FCC|                   ;
2FCC|                   ;   The cursor is a small image that is displayed on the screen.  It's shape
2FCC|                   ;   is specified by two bitmaps, called 'data' and 'mask'.  These bitmaps are
2FCC|                   ;   16 bits wide and from 0 to 32 bits high.  The rule used to combine the
2FCC|                   ;   bits already on the screen with the data and mask is
2FCC|                   ;
2FCC|                   ;       screen <- (screen and (not mask)) xor data.
2FCC|                   ;
2FCC|                   ;   The effect is that white areas of the screen are replaced with the cursor
2FCC|                   ;   data.  Black areas of the screen are replaced with (not mask) xor data.
2FCC|                   ;   If the data and mask bitmaps are identical, the effect is to 'or' the data
2FCC|                   ;   onto the screen.
```

```
2FCC|                       ;
2FCC|                       ;   The cursor has both a location and a hotspot.  The location is a position
2FCC|                       ;   on the screen, with X-coordinates of 0 to 720 and Y-coordinates of 0 to 364 .
2FCC|                       ;   The hotspot is a position within the cursor bitmaps, with X- and Y-coordi-
2FCC|                       ;   nates ranging from 0 to 16.  The cursor is displayed on the screen with it's
2FCC|                       ;   hotspot at location.  If the cursor's location is near an edge of the screen,
2FCC|                       ;   the cursor image may be partially or completely off the screen.
2FCC|                       ;
2FCC|                       ;-------------------------------------------------------------------------
2FCC|                       ;
2FCC|                       ;   Routine:    CursorInit
2FCC|                       ;   Arguments:  None
2FCC|                       ;   Function:   Sets up the initial defaults used by the ROM cursor.  Initial
2FCC|                       ;               position is set for center of the screen.
2FCC|                       ;
2FCC|                       ;-------------------------------------------------------------------------
2FCC|
2FCC| 4278 0490             CursorInit      MOVE.W  #0,CrsrHotX         ; cursor hotspot X-coordinate
2FD0| 4278 0492                             MOVE.W  #0,CrsrHotY         ; cursor hotspot Y-coordinate
2FD4| 31FC 0010 0494                        MOVE.W  #16,CrsrHeight      ; cursor hieght, 0-32
2FDA| 31FC 0168 0496                        MOVE.W  #360,CrsrX          ; initial cursor X-coordinate
2FE0| 31FC 00B6 0498                        MOVE.W  #182,CrsrY          ; initial cursor Y-coordinate
2FE6| 61CA                                  BSR.S   MousInit            ; init mouse for cursor control
2FE8| 4E75                                  RTS                         ; return
2FEA|
2FEA|                       ;------------------------------------------------------------------------
2FEA|                       ;
2FEA|                       ;   Cursor Hide and Display
2FEA|                       ;
2FEA|                       ;   Care must be taken when updating the screen image which is 'under' the
2FEA|                       ;   cursor.  The simplest approach is to remove the cursor from the screen
2FEA|                       ;   (hide), do the screen modification, then redisplay the cursor (display).
2FEA|                       ;   Each hide operation must be followed by a corresponding display
2FEA|                       ;   operation.  The operations are paired and can be nested.  The first of a
2FEA|                       ;   series of hides removes the cursor from the screen; it's corresponding
2FEA|                       ;   display redisplays the cursor.  Intervening operations have no apparent
2FEA|                       ;   effect.
2FEA|                       ;
2FEA|                       ;-------------------------------------------------------------------------
2FEA|
2FEA|                       ;----------------------------------------------------------------------------
2FEA|                       ;   Routine:    CursorHide
2FEA|                       ;   Arguments:  None
2FEA|                       ;   Function:   Remove the cursor from the screen.  Note that every call to
2FEA|                       ;               CursorHide must be followed by exactly one call to CursorDisplay.
2FEA|                       ;----------------------------------------------------------------------------
2FEA|
2FEA| 48E7 C0C0             CursorHide      MOVEM.L D0-D1/A0-A1,-(SP)   ; save registers
2FEE| 41F8 04A2                             LEA     SavedData,A0        ; saved data address
2FF2| 2278 0528                             MOVE.L  SavedAddr,A1        ; saved data screen address
2FF6| 3038 0526                             MOVE.W  SavedRows,D0        ; rows of saved data
2FFA| 323C 005A                             MOVE.W  #MaxX/8,D1          ; bytes per row on screen
2FFE| 6004                                  BRA.S   @2                  ; test of rows=0
3000|
3000| 2298                  @1              MOVE.L  (A0)+,(A1)          ; from saved to screen
3002| D2C1                                  ADD.W   D1,A1               ; screen address of next row
3004| 51C8 FFFA             @2              DBF     D0,@1               ; loop 'SavedRows' times
3008|
3008| 4CDF 0303                             MOVEM.L (SP)+,D0-D1/A0-A1   ; restore registers
300C| 4E75                                  RTS                         ; return
300E|
300E|                       ;-------------------------------------------------------------------------
300E|                       ;
300E|                       ;   Routine:    CursorDisplay
300E|                       ;   Arguments:    none
300E|                       ;   Function:   Redisplay the cursor.  Note that every call to CursorDisplay
300E|                       ;               must be preceeded by exactly one call to CursorHide.
300E|                       ;
300E|                       ;   Register Assignments:
300E|                       ;
300E|                       ;       D0  --  saved data X-coordinate, cursor data
300E|                       ;       D1  --  saved data Y-coordinate, cursor mask
300E|                       ;       D2  --  left shift count
300E|                       ;       D3  --  32-bit mask
300E|                       ;       D4  --  rows of saved data
300E|                       ;       D5  --  bytes per row on screen
300E|                       ;
```

```
300E|                     ;       A0  --  saved data address
300E|                     ;       A1  --  saved data screen address
300E|                     ;       A2  --  cursor data address
300E|                     ;       A3  --  cursor mask address
300E|                     ;-----------------------------------------------------------------------
300E|
300E| 48E7 FCF0           CursorDisplay   MOVEM.L D0-D5/A0-A3,-(SP)   ; save registers
3012|
3012| 41F8 04A2                           LEA     SavedData,A0       ; saved data address
3016| 2278 0110                           MOVE.L  ScrnBase,A1        ; screen memory address
301A| 45FA 0900                           LEA     CrsrData,A2        ; cursor data bitmap address
301E| 47FA 08FC                           LEA     CrsrMask,A3        ; cursor mask bitmap address
3022| 3038 0490                           MOVE.W  CrsrHotX,D0        ; cursor hotspot X-coordinate
3026| 3238 0492                           MOVE.W  CrsrHotY,D1        ; cursor hotspot Y-coordinate
302A| 3838 0494                           MOVE.W  CrsrHeight,D4      ; cursor height
302E|
302E|                     ;   Compute and bounds check the X-coordinate of the data under the cursor.
302E| 9078 0496           @11             SUB.W   CrsrX,D0           ;  cursor X-coordinate
3032| 4440                                NEG.W   D0                 ;  - cursor hotspot X-coordinate
3034| 3400                                MOVE.W  D0,D2              ; upper left X-coordinate
3036| 0242 000F                           AND.W   #$000F,D2          ; bit offset within word
303A| 4442                                NEG.W   D2                 ;  negated and converted to
303C| 0642 0010                           ADD.W   #16,D2             ;  left shift count
3040| 4283                                CLR.L   D3                 ; 32-bit mask                      RM000
3042| 4643                                NOT     D3                 ; D3 = $0000FFFF                   RM000
3044| E5AB                                LSL.L   D2,D3              ;  shifted into position
3046|
3046| 0240 FFF0                           AND.W   #$FFF0,D0          ; upper left X-coord rounded down
304A| 6C0A                                BGE.S   @0                 ; branch if >= 0
304C|
304C| 4240                                MOVE.W  #0,D0              ; minimum upper left X-coord of 0
304E| E18B                                LSL.L   #8,D3              ; adjust 32-bit mask
3050| E18B                                LSL.L   #8,D3              ; adjust 32-bit mask
3052| 0642 0010                           ADD.W   #16,D2             ; adjust left shift count
3056|
3056| 0C40 02B0           @0              CMP.W   #MaxX-32,D0        ; upper left X-coord <= 720-32
305A| 6F14                                BLE.S   @2                 ; branch if <= 720-32
305C|
305C| 0C40 02D0                           CMP.W   #MaxX,D0           ; cursor off right edge ?
3060| 6602                                BNE.S   @1                 ; branch if not off right edge
3062| 7600                                MOVEQ   #0,D3              ; mask off all bits
3064|
3064| 303C 02B0           @1              MOVE.W  #MaxX-32,D0        ; maximum X-coord of 720-32
3068| E08B                                LSR.L   #8,D3              ; adjust 32-bit mask
306A| E08B                                LSR.L   #8,D3              ; adjust 32-bit mask
306C| 0642 0010                           ADD.W   #16,D2             ; adjust left shift count
3070|
3070|                     ;   Compute and bounds check the Y-coordinate of the data under the cursor.
3070|
3070| 9278 0498           @2              SUB.W   CrsrY,D1           ; cursor Y-coordinate
3074| 4441                                NEG.W   D1                 ;  - cursor hotspot Y-coordinate
3076| 6C0C                                BGE.S   @3                 ; branch if upper left Y >= 0
3078|
3078| D841                                ADD.W   D1,D4              ; decrease rows of saved data
307A| D241                                ADD.W   D1,D1              ; double for byte count
307C| 94C1                                SUB.W   D1,A2              ; increase cursor data address
307E| 96C1                                SUB.W   D1,A3              ; increase cursor mask address
3080| 4241                                MOVE.W  #0,D1              ; minimum upper left Y of 0
3082| 6010                                BRA.S   @4                 ; continue
3084|
3084| 3A3C 016C           @3              MOVE.W  #MaxY,D5           ; maximum Y-coordinate
3088| 9A44                                SUB.W   D4,D5              ;  - cursor height
308A| B245                                CMP.W   D5,D1              ; cursor bottom <= 364-CrsrHeight ?
308C| 6F0C                                BLE.S   @5                 ; branch if <= 364-CrsrHeight
308E|
308E| 383C 016C                           MOVE.W  #MaxY,D4           ; last row on screen
3092| 9841                                SUB.W   D1,D4              ; adjust rows of saved data
3094|
3094| 4A44                @4              TST.W   D4                 ; rows of saved data >= 0 ?
3096| 6C02                                BGE.S   @5                 ; branch if  >= 0
3098| 4244                                MOVE.W  #0,D4              ; minimum rows of saved data
309A|
309A| 31C0 0522           @5              MOVE.W  D0,SavedX          ; saved data X-coordinate
309E| 31C1 0524                           MOVE.W  D1,SavedY          ; saved data Y-coordinate
30A2| 31C4 0526                           MOVE.W  D4,SavedRows       ; rows of saved data
30A6|
30A6|                     ;   Display the cursor on the screen.
```

```
30A6|
30A6| E648                         LSR.W   #3,D0              ; convert X-coord to bytes
30A8| D2C0                         ADD.W   D0,A1              ;  and add to screen address
30AA| 7A5A                         MOVEQ   #MaxX/8,D5         ; bytes per row on screen
30AC| C2C5                         MULU    D5,D1              ;  * Y-coord
30AE| D3C1                         ADD.L   D1,A1              ;  added to screen address
30B0| 21C9 0528                    MOVE.L  A1,SavedAddr       ; saved data screen address
30B4| 6016                         BRA.S   @7                 ; test for rows=0
30B6|
30B6| 301A              @6         MOVE.W  (A2)+,D0           ; cursor data
30B8| E5B8                         ROL.L   D2,D0              ; shift to proper bit position
30BA| C083                         AND.L   D3,D0              ; eliminate unwanted bits
30BC| 321B                         MOVE.W  (A3)+,D1           ; cursor mask
30BE| E5B9                         ROL.L   D2,D1              ; shift to proper bit position
30C0| C283                         AND.L   D3,D1              ; eliminate unwanted bits
30C2| 4681                         NOT.L   D1                 ; invert cursor mask
30C4|
30C4| 20D1                         MOVE.L  (A1),(A0)+         ; from screen to saved data
30C6| C391                         AND.L   D1,(A1)            ; screen and (not mask)
30C8| B191                         EOR.L   D0,(A1)            ;  xor cursor data
30CA| D2C5                         ADD.W   D5,A1              ; screen address of next row
30CC| 51CC FFE8         @7         DBF     D4,@6              ; loop 'SavedRows' times
30D0|
30D0| 4CDF 0F3F                    MOVEM.L (SP)+,D0-D5/A0-A3  ; restore registers
30D4| 4E75                         RTS                        ; return
```

Lisa Schematics
Apple Computer, 1983-1984, 25 pages

These schematics can be useful if the hardware manual is unclear about how Lisa subsystems are designed and how they communicate.

SYSTEM.LLD Disassembly
David T. Craig, 1987-1988, 63 pages

This document (which consists of 2 documents) provides lots of information about the Lisa memory-mapped I/O memory locations. It describes the low-level drivers (LLD) which the Lisa OS and its drivers call when they want to interface with the Lisa hardware. This interface controls the mouse, screen location, screen contrast, speaker, keyboard, alarms, clock, disk drives.

Lisa Workshop development manuals
Apple Computer

This series of 3 manuals describes in detail how to write programs for the Lisa. Though much of this information will not help the emulator writer this series does provide lots of great background information on how the Lisa works. The 3 manuals in this series are:

o	**Lisa Language**	**(453 pages)**
o	**Lisa Workshop User's Guide**	**(399 pages)**
o	**Lisa System Software**	**(352 pages)**

The Lisa Language manual (which describes the Pascal language) contains in description of the Lisa hardware library in Appendix F which any emulator writer would need to know very well. The interface for this library follows:

```
type

Pixels          = Integer;
ManyPixels      = LongInt;
CursorHeight    = Integer;
CursorPtr       = ^Integer;
DateArray       = Record
                    year  : Integer;
                    day   : Integer;
                    hour  : Integer;
                    minute: Integer;
                    second: Integer;
                  end;
Frames          = LongInt;
Seconds         = LongInt;
MilliSeconds    = LongInt;
MicroSeconds    = LongInt;
SpeakerVolume   = Integer;
ScreenContrast  = Integer;
KeybdQIndex     = 1..1000;
KeybdId         = Integer;
KeyCap          = 0..127;
KeyCapSet       = Set of KeyCap;
KeyEvent        = Packed Record
                    key   : KeyCap;
                    ascii : Char;
                    state : Integer;
                    mouseX: Pixels;
                    mouseY: Pixels;
                    time  : MilliSeconds;
                  end;

{ Mouse }

Procedure MouseLocation (var x: Pixels; var y: Pixels);
Procedure MouseUpdates (delay: MilliSeconds);
Procedure MouseScaling (scale: Boolean);
Procedure MouseThresh (threshold: Pixels);
Function  MouseOdometer: ManyPixels;

{ Cursor }

Procedure CursorLocation (x: Pixels; y: Pixels);
Procedure CursorTracking (track: Boolean);
Procedure CursorImage (hotX: Pixels; hotY: Pixels; height: CursorHeight;
                       data: CursorPtr; mask: CursorPtr);

Procedure BusyImage (hotX: Pixels; hotY: Pixels; height: CursorHeight;
                     data: CursorPtr; mask: CursorPtr);
Procedure BusyDelay (delay: MilliSeconds);

{ Screen }

Function  FrameCounter: Frames;
Procedure ScreenSize (var x: Pixels; var y: Pixels);
Function  Contrast: ScreenContrast;
```

```
  Procedure SetContrast (contrast: ScreenContrast);
  Procedure RampContrast (contrast: ScreenContrast);
  Function  DimContrast: ScreenContrast;
  Procedure SetDimContrast (contrast: ScreenContrast);
  Function  FadeDelay: MilliSeconds;
  Procedure SetFadeDelay (delay: MilliSeconds);

  { Speaker }

  Function  Volume: SpeakerVolume;
  Procedure SetVolume (volume: SpeakerVolume);
  Procedure Noise (waveLength: MicroSeconds);
  Procedure Silence;
  Procedure Beep (waveLength: MicroSeconds; duration: MilliSeconds);

  { Keyboard }

  Function  Keyboard: KeybdId;
  Function  Legends: KeybdId;
  Procedure SetLegends (id: KeybdId);
  Function  KeyIsDown (key: KeyCap): Boolean;
  Procedure KeyMap (var keys: KeyCapSet);
  Function  KeybdPeek (repeats: Boolean; index: KeybdQIndex;
                     var event: KeyEvent):
                     Boolean;
  Function  KeybdEvent (repeats: Boolean; wait: Boolean; var event: KeyEvent):
                      Boolean;
  Procedure RepeatRate (var initial: MilliSeconds;
                      var subsequent: MilliSeconds);
  Procedure SetRepeatRate (initial: MilliSeconds; subsequent: MilliSeconds);

  { Timers }

  Function  MicroTimer: MicroSeconds;
  Function  Timer: MilliSeconds;

  { Date and Time }

  Procedure DateTime (var date: DateArray);
  Procedure SetDateTime (date: DateArray);
  Procedure DateToTime (date: DateArray; var time: Seconds);

  { Time Stamp }

  Function  TimeStamp: Seconds;
  Procedure SetTimeStamp (time: Seconds);
  Procedure TimeToDate (time: Seconds; var date: DateArray);
```

The Lisa Workshop User's Guide contains a section (chapter 8) on LisaBug, the Lisa's low-level 68000-oriented debugger. This volume also describes all of the Workshop's utility programs (e.g. the 68000 code file disassembler).

Lisa Chip Data Sheets
40 pages or so

These are the technical data sheets which chip manufactures created. I have the data sheets for the VIA, COPS, and the AM9512 math chip (a socket for this is available on Lisa 1 machines, but the Lisa software never really supported this chip to my knowledge so I recommend that this chip not be supported by the emulator). These specs are needed so you will understand the gory details behind these chips.

ImageWriter Printer Reference Manual
Apple, 1984

This was the printer that many Lisa user's used. This printer supported 3 modes of printing: draft (used the printer's built-in character sets, ugly but fast), standard (72 dpi bitmapped output, ok looking, somewhat slow), and high quality (144 dpi, used for most printing needs, very good looking, slow). For a successful emulator I recommend that this printer's output be emulated. To do this you would need to write a "driver" for this printer within the emulator that emulated all of the printer's control codes. The output of this driver should be a file which the host computer could print. I recommend that the output of this driver be a generic file containing a set of bitmapped pages at 72 or 144 dpi. A separate program residing on the host computer would then look for these files and print them to the host machine's printer. This strategy removes printing details from the emulator and assigns it to another program which can be modified by itself without effecting the emulator.

David Craig also has several printouts from the ImageWriter for the Lisa showing what the Lisa could do. David also has a complete listing of all of this printer's native built-in character sets which an emulator may want to support for Lisa draft printing.

68000 Assembly Language Programming
Kane, Hawkins, Leventhal, 1981

This is a great 68000 book. I recommend this book and Motorola's original 68000 book (it has a blue cover).

Microcomputer Architecture and Programming: The 68000 Family
John F. Wakerly, 1989

This is a good book that compliments Leventhal's 68000 book. Includes some fairly detailed information about 68000 emulation using the Pascal language (see page 155+).

MacWorks Plus: Making a Lisa Speak Macintosh
Charles Lukaszewski, Sun Remarketing, 1989, 4 pages

This article describes how the Macintosh emulator that ran on a Lisa was created. It provides some good information about the low-level disk booting aspects of the Lisa's boot ROM.

The Legacy of the Apple Lisa Personal Computer: An Outsider's View
David T. Craig, 1993, 19 pages

This paper attempts to describe the key aspects of the Lisa's operation and architecture. It contains several pictures of the Lisa including a sample Lisa screen showing its GUI interface. It also contains an extensive reference section listing the majority of the 300+ Lisa documents I have and which others may want to read. These include the Lisa Product Introduction Plan (PIP) and the Lisa Marketing Requirements Document (MRD).

That's all folks!