Emulating the Fred Machine

Brian P. Foley

Master of Science
School of Informatics
University of Edinburgh
2003

Abstract

The APM, or Fred Machine, was an experimental networked workstation designed by the University of Edinburgh Department of Computer Science in the early 1980s. Development continued throughout the 1980s, with many hardware components and a large collection of software being added until the system was decommissioned in the early 1990s.

Even though only 60 or so of the machines were ever built[5], it is instructive to examine the design decisions behind both the hardware and software. As a whole, the system has some interesting strengths and weaknesses, and in some ways was very much ahead of its time. In 2001, as a 4th year project, Cristopher Lisle wrote a partial emulation of the Fred Machine and the networked filestore. This project extends this work by improving the emulation and by researching in further detail the hardware and software that made up the system.

Acknowledgements

John Butler for providing documentation, advice, and software from the Fred Machine.
George Ross for answering far too many questions on how the Fred Machine internals really worked.

Christopher Lisle for the original Fred Machine emulation.

Karl Stenerud for the M68000 emulator core.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Brian P. Foley*)

Table of Contents

Chapter 1

Introduction

Yet herein will I imitate the sun,
Who doth permit the base contagious clouds
To smother up his beauty from the world,
That when he please again to be himself,
Being wanted, he may be more wondered at

William Shakespeare, *Henry IV*

1.1 What is the Fred Machine?

In the late 1970s and early 1980s, it was observed that if Moore's Law[1] continued to hold, it would soon be possible to put astonishingly powerful workstations on the average researcher's desk. One oft talked about milestone was the *3M Machine*: an affordable workstation with 1MB of memory, a 1 MIPS CPU, and a 1 megapixel display. The Fred Machine, built in Edinburgh, more than meets all of these design targets.

As well as this, the Fred Machine has other desirable attributes. Being based on the locally developed *Fred Bus*, it is very modular and flexible. From the beginning it was designed to be a networked workstation, and using a smart network adapter which could handle low level networking protocol details, it communicates over a 2.1Mb/s Ethernet-like network.

[1] An empirical observation that the number of components that can be fitted onto a given area of silicon at a given price point seems to double approximately every 18 months

The hardware and software for the Fred Machines were built up and refined over the course of the 10 or so years that they were in operation. Many staff and students wrote a variety of software for the system, including compilers, assemblers, interpreters, computer vision software, circuit design software, graphics libraries, games, a TEX environment, and even new operating systems.

Much of the design work for the core Fred Machine system was documented in a set of reports and technical documentation, and much of this still exists. A number of backups of some of the network filestores have been kept on CD.

This project aims to write software to emulate on a modern computer the behaviour of the Fred Machine and the network it ran on. This will allow the original Fred Machine software to be 'brought back to life' and to be run inside this simulated environment. Since a previous project[1] attempted to emulate the Fred Machine hardware with some degree of success, this project follows on from there.

Before giving more detailed goals for this project, we first give the reader an overview of the Fred Machine and its history.

1.2 The History of the Fred Machine

The Fred Machine project was started in 1981 with the initial intent of being a flexible platform for experimenting with computer hardware. The hope was that the Computer Science Department could extend their expertise with computer software and theoretical computer science to include the hardware side of computing.

A number of computer systems developed in Edinburgh in the late 1970s had a strong influence on the development of the Fred Machine project.

1.2.1 Background

1.2.1.1 ELAN

One of the most obvious influences was the development of Ethernet at Xerox PARC[8] in the mid-1970s. This work prompted interest in Edinburgh, and two experimental network designs were explored. By 1980, an Edinburgh team had agreed on one of

the designs, and went about setting up a network called ELAN (Edinburgh LAN)[3]. This was based on a system that was broadly similar to Ethernet, that is to say, it was based on a shared bus and collision detection hardware. Since Ethernet had not formally been standardised at this point, some details were different–the frame format differed, timings and the maximum network size differed, and most significantly it ran at 2.1Mb/s rather than 10Mb/s.

The ELAN boards, or *Stations*, became a core part of the Fred Machine.

1.2.1.2 The Filestore

Within the Computer Science Department in the late 1970s a number of Interdata series 70 minicomputers were available and ran the locally developed ISYS operating system. When a number of discless Interdata 74s were introduced, the filestore–a networked fileserver on the Interdata 70 was written[4] to allow them to share expensive storage. The filestore was based on LEGOS, another Edinburgh OS based on ISYS.

1.2.1.3 IMP and EMAS

EMAS (the Edinburgh Multi-Access System) was a time-sharing operating system written for large mainframes in Edinburgh in the 1970s[2]. As with UNIX, most of it was written in well structured code in a high level language. In this case, the language was IMP, a language designed in Edinburgh which was derived from Atlas Autocode.

The design of EMAS influenced the Fred Machine, and one of the first things to be done on the software side of the Fred Machine project was to write a reliable IMP cross compiler and a native IMP runtime for the Fred Machine itself. Since the IMP cross compiler was also written in IMP, it could be compiled on itself. Because of this, as soon as the runtime was complete, the Fred Machine had a fully functioning self-hosted compiler. After this, much of the Fred Machine software was written in IMP.

[2]EMAS ran on a number of ICL machines throughout the 1970s and early 1980s, including a 4/75, followed by a 2972, a 2976, and a 2980.

1.2.2 The Fred Bus

One of the main features directly attributable to the Fred Machine project was the development of the Fred Bus. The Fred Bus was designed to be a simple and flexible bus that made it easy to build add-on boards, whilst still being reasonably powerful. It has a 32 bit address space, and is capable of transferring 8, 16 or 32 bit words. Devices on the bus can act as either masters or slaves, and there is a well defined protocol for acquiring the bus, and determining which card should be the bus master.

1.2.3 The First Fred Machines

The first 3 prototype Fred Machines were built between 1981 and 1982, by a group of three people; Fred King, Hamish Dewar, and Rainer Thonnes.

Initially, it had just one card on the bus–an 8MHz M68000 with 16KB of local memory (divided into a boot ROM and some RAM), a Motorola 6840 providing programable timers, a Motorola 6850 providing an RS-232 port, and a simple MMU for translating memory accesses by the M68000 into Fred Bus transactions.

Once this was functioning, the next card to be added was one containing a ½MB of RAM. Since the location where these appeared in the Fred Bus address space was configurable by hex switches on the card, several of these could be added, allowing the CPU to access up to 7½MB of external RAM.

The most sophisticated card to appear on the prototype machines was an ELAN Station. This has hardware which can send and receive frames on the ELAN network. The hardware is controlled by a Z80 processor with firmware in EPROM and 4KB RAM. The firmware handles the low level mechanics of frame transmission, as well as a managing buffering and acknowledgement of packet receipt. The station provides a number of registers allowing it to communicate directly with the CPU card.

On the software side, a 68000 IMP cross-compiler as well as an assembler and disassembler all written in IMP, were written and run on the Interdata machines. Since there was no operating system on the Fred Machine yet, the IMP runtime had to be written to work directly with the bare hardware, and had to provide some simple operating system services. Since these development tools were written in IMP, once they

were sufficiently complete, the compiler could compile itself, and thus a self-hosting development system was created for the Fred Machine in short order. When this had been done, further development could continue on the Fred Machines. This helped the development process, since the Fred Machines had much more memory, and were faster than the Interdatas[3].

1.2.4 The APM

Once the work on the prototype Fred Machines had been completed, the Fred Machines had become a useful, if somewhat bare bones, workstation.

In the summer of 1982, the workstation was demonstrated at a University Open Day, and was introduced as the *Advanced Personal Machine*. As the name implies, this meant that, rather than being primarily a research platform, the Fred Machine was now targeted at being a general purpose workstation to be used by staff and students. To some extent, this new emphasis constrained the further development of the Fred Machine.

However, in both software and hardware, a number of advances were made over the prototype machines.

The memory boards were revamped to take advantage of denser DRAM chips, and so were able to hold 2MB RAM rather than ½MB.

The CPU on the CPU board was upgraded to the M68010 with MMUs, and this made virtual memory possible, although the operating system never took advantage of it.

A number of video boards were produced, the simplest of which provided a 1024 × 1024 pixel framebuffer, with either 4 bits per pixel, or when upgraded, an 8-bpp framebuffer with a programable colour lookup table.

A hard disc controller and laster printer controller were built, which meant that a Fred Machine system could replace an Interdata filestore as a file and print server.

[3] The Interdata 74 was fitted with 32KB of memory, for example.

1.3 Project Resources

To emulate a computer system three things are needed: Detailed specifications on the hardware components; information on how they are interconnected; and software (ideally with documentation) to run on the system.

Thankfully, for the most part, all the above existed for this project.

- The three most complex chips used in the Fred Machine are the Motorola M68010 on the CPU board, the Zilog Z80 on the ELAN Station, and the 6809 on the 6809 kit board. All of these are very well known, well documented chips, and many emulators have been written for them[4].

- The interconnects are unique to Edinburgh, but fortunately they are both simple (from a software point of view), and partially documented.

- The Fred Bus specification describes operations at a very low level, something which is appropriate for an electronic engineer. This provides us with more information than we need for an emulation[5]. For a correctly functioning emulator, we can impose simplifying assumptions[6], and thus can ignore most of the intricacies of the Fred Bus protocol.

- The custom Fred Bus MMU on the CPU board, and the I/O peripheral addressing provided by the board are also well documented[6]. These are also implemented by Christopher Lisle's project, which provides a useful compliment to the specifications.

- The interface that the Level 1 and Level 1½ graphics boards expose to the Fred Bus is documented in the working documents[6]. A minor error in this specification was discovered when trying to test real graphics software, but this was easily fixed.

---

[4] As an example, see the emulator cores in the MAME project[10]

[5] Of course having too much information is a much better position to be in than having too little information.

[6] Such as pretending bus transactions are instantaneous, and thus cannot interfere with each other.

- The original ELAN network and the ELAN Station are also well documented[3]. A couple of details are unspecified, such as what the generating polynomial for the 16 bit CRC checksum is.[7] This is unimportant for a purely software emulation, as we can create our own frame format, but it does become an issue if we want to interface with real ELAN hardware.

  Unfortunately, although the ELAN Stations used in the Fred Machines are based on the original ELAN work, they have a completely different (if conceptually similar) interface to the CPU board. Some details of the host/station protocol which passes over this interface have also changed. These changes were primarily motivated, it appears[8] by a need to allow the filestore to have more than 15 users logged on simultaneously. We were unable to find any explicit definition or documentation of the interface or protocol changes, and so had to resort to analysing source code from the Fred Machine to infer the design. Happily, a major amount of the work done on the original Fred Machine Emulation involved determining how the networking worked, and writing a partly functional emulation, and much of this can be reused.

- The ELAN Station doesn't just send and receive frames, it also implements a higher level protocol. From an emulation point of view, it is unfortunate that it does this. The protocol, while explained in the ELAN document, is to a certain degree open to interpretation. A state transition diagram, or something similar would have been useful.

- The physical interface the ELAN Station uses to communicate with the CPU reveals another problem. There is a second bus used on the Fred Machines called the *Front Bus*. This appears to be an electrical extension of the Local Bus on the CPU board, and allows the CPU to read and write to other cards' registers, and allows the cards to assert interrupts on the CPU. No documentation was found for this bus, and as it is used for the ELAN Station, the DMACK serial controller, and other cards such as the 6809 kit board, and so must be emulated, we must

[7] It cannot be the same as standard Ethernet, as it has a 32 bit checksum. At a guess, it is probably CRC16-CCITT.

[8] Personal communication with George D.M. Ross

guess at its behaviour.

- There is an extensive document describing the original filestore[4], however there are a few problems with it. Firstly, the protocol itself has been subject to revision over the years. New features have been added, and behaviour changed. See, for instance, [7]. Secondly, the precise semantics of the filestore protocol are not always described. For many commands, the document does not describe exactly how failures should be handled, and what particular errors should be reported if they do. The source of a number of versions of the filestore (written in IMP) were made available for this project, and this helped clarify many details on how the filestore was expected to behave.

- On the software side, a number of extensive backups of a number of the filestores from 1987, 1989, 1991 and 1993 were made onto CD-ROM, totalling several hundred megabytes of source code, binaries, and data. Excerpts from these have been made available for testing the emulation. A useful emulation would have been close to impossible without them. The availability of source code makes debugging the emulation considerably easier. It is much easier to infer the propose of a program, and how it works, from the source, rather than having to disassemble a binary object.

1.4 Project Goals

Thanks to the Fred Bus, and the fact that to a certain extent the Fred Machine was regarded as a research machine, there is no one canonical Fred Machine with a well defined set of peripherals. Thus, to do an emulation, some design tradeoffs must be made. Some of these choices are mutually exclusive, some are not. For example, do we want to emulate Level 1 or Level 2 graphics? Do we want to emulate a M68010 or a M68030 processor[9]? Do we want to emulate multiple processor boards? How about the DMACK boards, the hard disc controller, and the laser printer controller?

[9] On an OS level they require mutually incompatible exception handling. Most of the Fred Machines used 68010s.

Another question, which arises with all emulators, is at what level do we want to emulate various pieces of functionality? For instance, we need to emulate the Ethernet station, since it is central to the system, but at what level do we emulate–emulate the low level hardware, and run the firmware on a Z80 emulator? Or perhaps emulate the behaviour based on the interface exposed to the CPU? We could even try to trap the calls at a higher level in the operating system (perhaps by patching the OS kernel when it is loaded).

A third, and potentially quite complex, question is how do we interface with the host environment? As a case in point, nobody uses 2.1Mb/s Ethernet anymore, so how are we going to transfer frames from one emulated Fred Machine to another? Or from an emulated Fred Machine to a real one? Or, for that matter, from one emulated Fred Machine to software in the host environment? Similar questions arise for terminal I/O, graphics emulation, hard disc emulation, and mouse input.

The overriding design decision for this project can be summed up with the term *Software Archæology*. That is, while the hardware may be interesting in its own right, for the purposes of this project we only care about it insofar as it provides a platform on which to run existing software. Thus, if none of the existing software base ever used a particular feature of the Fred Machine hardware, or if emulating a particular detail of the hardware unduly complicates or slows the emulation without having much of an effect on the software, then we ignore it.

Some of these decisions are detailed in chapter 5.

Since the previous project got to the stage of providing a read-only UNIX filestore, a partly functioning network simulation, could boot the simulated Fred Machine, and could run a few simple commands before failing, we must extend on this work.

A primary target was chosen for this project: to improve the emulation to the point where we can run interactive graphical programs.

1.5 The Results, in Brief

The project was successful in meeting its target.

- The behaviour Level 1½ graphics card has been accurately emulated; a number

of fixes were made to the networking and filestore, to make it more robust in the face of an interactive environment; and quite a bit of graphical software was found in the system backups and was coaxed into to working.

- Level 1 graphics test programs such as `TESTCARD`, and `BARPOLE` work. Lots of graphics demos such as `PERSIAN`, `HARMONY`, and `RUBIK` work.

- More sophisticated programs that use the `EDWIN` library, such as `SPIRO`, `VIEWPDF`, `GILVIEW`, and a vector graphics editor called `DRAW` run under emulation. Unfortunately `DRAW` requires a mouse for user input, and an emulation of this hardware wasn't completed at the time of writing.

- One of the most demanding tests was to get interactive games to run. `FROGGER` runs completely successfully. Three other games–`PACMAN`, `AST`[10], and `TAILGUN` all work, but since they need a mouse for user input, they aren't playable at the time of writing.

[10] An Asteroids clone

Chapter 2

Fred Machine Internals

We are what we pretend to be, so we must be careful of what we pretend to be.

Kurt Vonnegut, *Mother Night*

Half the work that is done in the world is to make things appear what they are not

---

Elias Root Beadle

2.1 Fred Machine Hardware

2.1.1 The Cabinet

The Fred Machine hardware is designed, both logically and physically, as a set of boards hooked up to a common bus. The Fred Machine cabinet (CSD144 described in [6]) is a metal case, with a perspex door and a 400W power supply. It is frequently described as being microwave like in appearance. It has a backpane that can hold up to 19 Fred Bus cards, and provides pins to arbitrate bus mastering requests from individual cards.

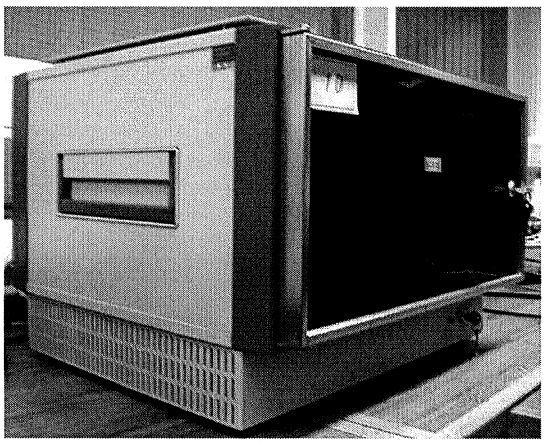

Figure 2.1: The Fred Machine case

2.1.2 The Fred Bus

The Fred Bus is a medium-speed 32 bit bus connecting intelligent peripherals together. Cards attach to the bus via a 96 pin connector. The pins provide power, bus signalling, a 100Hz clock, and 32 data and 32 address lines.

The cards can operate in one of two modes– *Master* or *Slave*, and there is a 3 level protocol for defining how cards communicate with each other.

The first step in performing a Fred Bus transaction involves taking control of, or acquiring the bus. Obviously, it would a bad thing for multiple devices to attempt to send data simultaneously. A card numbered i which isn't already in control of the bus requests control by asserting the BRQi (Bus Request) pin, and waits for the bus controller to grant it by asserting the BGRi (Bus Grant) pin. This will not occur until other cards finish their transactions. The BRQi pin remains asserted for the duration of the transaction.

The second step involves the bus master specifying a destination address and sending an 8, 16 or 32 bit data transfer across the bus. Each of the slave cards listens for transfers, and if the address matches a card, it responds by performing the appropriate read or write, and can either acknowledge the transfer, or raise an error.

For the third and final step, the bus master receives the acknowledgement or error, and can read any resulting data (if necessary), and finally releases the bus.

As we can see this procedure is rather involved, and it has to be done for every read

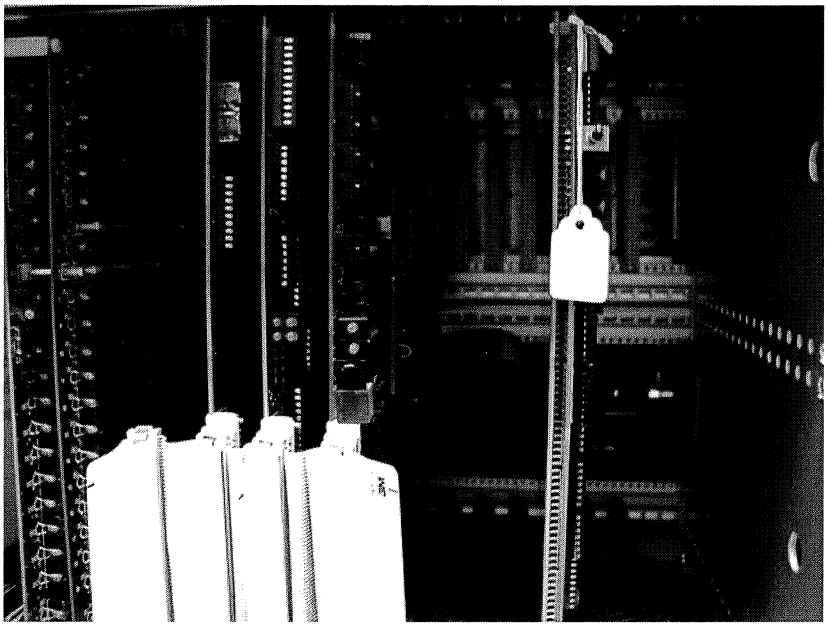

Figure 2.2: Inside the Fred Machine

or write to system memory, for every update to the video display, and so on.

2.2 The Arbitration Board

On most Fred Machines, there is only ever one card (the CPU board) that can initiate transactions on the Fred Bus. In this particular situation, there is no need to have any hardware to check whether a device already has control of the bus–the CPU board just asks for control of the bus, and is immediately granted it. In hardware, this is done by connecting the BGR and BRQ lines together, so as soon as a request is made, it is immediately granted.

Some more complex Fred Machines were built with multiple processor cards. Since each of the CPUs could individually start transactions, bus arbitration hardware needed to be present to handle this. The hardware took the form of an extra Fred Bus card (that used the Fred Bus simply for power lines). The BGR and BRQ pins from each of the CPU cards were connected directly by wires to a set of pins on the arbitration board.

According to George Ross, the Fred Bus itself was a relatively noisy bus, electrically speaking (since there were so many pins in such a confined space), and there were subtle bugs in the arbitration board. This caused problems with the reliability of the multiprocessor systems.

It is interesting to note that a without the arbitration board, similar details of the Fred Bus transaction sequence are ignored as are by the emulator. This suggests that this is not an unreasonable approach to take. The only difference is that an emulator can guarantee in code that bus collisions cannot occur, whereas in hardware we must take care not to use the wrong mixture of cards.

2.3 The Processor Board

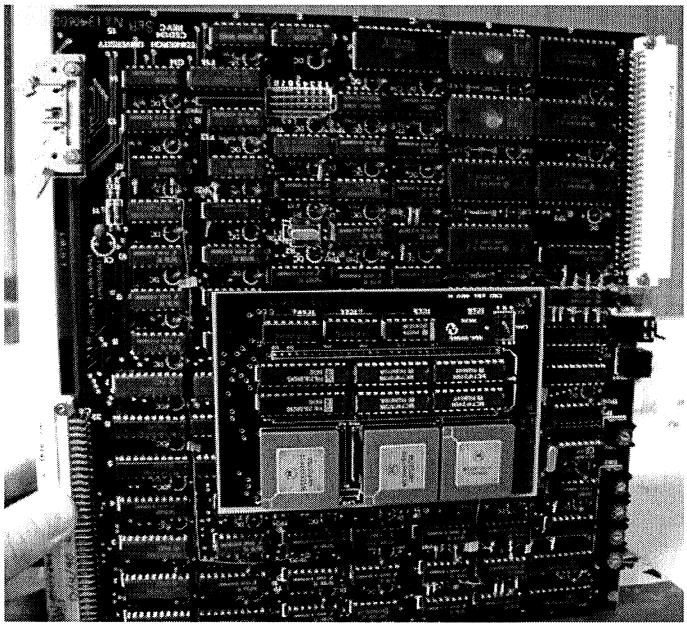

Figure 2.3: The CPU board

In one sense, the CPU is the 'heart' of the Fred Machine. Without something to initiate and coordinate the activities of the system, all the cards will just sit idle waiting for transactions to happen.

That said, from a Fred Bus point of view, there is nothing special about the Processor Board–it is just another board on the bus, albeit one which starts rather a lot of bus transactions. One consequence of this is that, arbitration hardware aside, there is no special hardware required to have multiple CPU boards in the one system. Special software has to be designed to allow the boards to perform meaningful work together, but the hardware works as is.

2.3.1 The CPU

The CPU board in most Fred Machines holds a small daughter-board with 10MHz Motorola M68010 microprocessor and a pair of Motorola 68451 MMUs. The 68451 was Motorola's first attempt at an MMU for the 680x0 processors, and apparently was dramatically inferior to the much more popular 68851 that Motorola introduced later. A pair of MMUs is required to make paging with 4K pages possible, although in practice, none of the standard Fred Machine software actually uses the MMU.

The daughter-board is plugged into a local bus on the CPU board, along with 16KB of local memory which is arranged in 4KB chunks of either ROM or RAM. This is usually configured with the first 4KB containing two 8-bit 2KB EPROMs used to bootstrap the system, and the remaining 12KB holding 6 8-bit 2KB DRAM chips as local RAM. The OS kernel is loaded into the local RAM, and things like the OS *pseudo-vector table* and various OS housekeeping information are stored here too. This is sensible, as the local memory is *fast memory* since accessing it doesn't require transactions to go across the Fred Bus.

The M68010 communicates with every component on the CPU board via the *Local Bus*. To access the Local Bus, the 68010 has 24 address pins (A0-A23), onto which it puts the address it wants, as well as another pin which specifies whether it wants to read or write, and a final set of pins which are used to specify the data that is read or written. Circuitry on the local bus routes the memory accesses to the correct components, thus providing both memory mapped I/O, and access to local and system memory. The particular mechanisms used are described later.

2.3.2 The M6800 Bus

Hooked up to this Local Bus is an M6800 bus, which uses M6800 timing, and signalling which is compatible with a number of Motorola's peripheral chipsets. Two of these chips are used on the CPU board: the Motorola 6840–a chip with 3 programmable timers, and a Motorola 6850 which is an RS-232 serial port controller.

The 6840 appears on the M6800 bus as 8 readable and writable single byte registers. Its interrupt pin is connected to an interrupt line on the Local Bus, and generates a level 6 interrupt on the CPU.

The serial controller appears as 2 single bytes registers on the M6800 bus. It is hooked up to a set of toggle switches which allow the user to choose the baud rate at which the serial port operates. Its interrupt pin is hooked up to the M68010 via a line on the local bus, and generates a level 5 interrupt on the CPU.

The other device on the M6800 bus is a rotatable mains switch which is used when the user powers on the system. It appears as a single byte read only register.

2.3.3 Interface with the Fred Bus

A final set of devices to appear on the Local Bus is a set of registers that control how the CPU generates Fred Bus transactions, and a set of registers that allows other boards on the Fred Bus to read the CPU state and generate programmable interrupts.

2.3.3.1 Address Mapping

The first set of registers forms a set of 8 *address mapping* registers. They consist of 12-bit write only registers that appear only on the Local Bus, and are used to translate the addresses the CPU specifies when accessing system memory into addresses suitable for the Fred Bus. The CPU can access 8MB of system memory, and each register maps a 1MB block of memory onto a 1MB block of the Fred Bus address space. Address lines A22-A20 are taken as a 3-bit index into the 8 entry address map table, and are used to generate the high order 12 bits of the Fred Address. The low 20 bits come from A19-A0.

Since the M68010 can't write 12 bit words, the registers are set by writing a 16 bit word. The top 12 bits are extracted from the write and put in the appropriate register.

As the address mapping registers are write only, the APM working guide recommends that whenever system software updates the registers, it should keep a readable copy of its contents in RAM.

2.3.3.2 Processor State

The second set of registers is accessed from the 'outside' by other cards on the Fred Bus, so they are not visible on the CPU's memory map. The Fred Bus addresses that these registers appear at are controlled by a set of 4 hex switches on the CPU board. These define the top 16 bits of the 64KB block of memory where the CPU board appears on the Fred Bus. Within this 64KB block are 9 single-byte registers, which are selected with bits 15-13 of the Fred address.

Register 0 is a read only Processor State Register. Bits 0 and 1 are always set to 1; bits 2 and 4 are always set to 0; bit 3 is set if the processor halts after a fatal bus error; and bits 5, 6, and 7 hold the 3 bits of the function code for the last CPU cycle.

Registers 1-7 are interrupt registers–when read they return an interrupt state register (which has bit 7 set if there is an interrupt pending at the given interrupt level), and when any value is written to them, they notify the CPU of an interrupt, and set the appropriate bit of the interrupt register.

When the CPU acknowledges an interrupt, the corresponding interrupt register is cleared.

The final register is called the Combined Interrupt State Register, is read only, and contains the state of the other 7 interrupt registers. Bits 1-7 of the interrupt register are set to 1 if there is a level 1-7 interrupt pending. Again, when the CPU acknowledges an interrupt, the corresponding bit in the CISR is cleared.

2.3.4 The Memory Map

The memory map of the CPU board is formed by circuitry on the board which decodes different address lines, and uses them as a switch to access different components on the board.

2.3.4.1 System Addresses

The main switch is the A23 line. When this is 0, the access is to the Local Bus. When 1, it is to the System Bus[1]. Thus the first 8MB of memory is local, and the 2nd 8MB of memory is mapped onto Fred Bus accesses. Further mapping of system addresses occurs as described in section 2.3.3.1.

2.3.4.2 Address Mapping Registers

Local addresses are decoded further by examining A19. When this is set to 1, the addresses A22-A20 define which of the 8 address map registers should be written to. This particular arrangement means that the local memory map is divided up into 16 512KB chunks, and that writing to any word in one of the odd numbered chunks (where the 1st chunk is numbered 0) updates the appropriate address mapping register.

2.3.4.3 The Local Bus

If A19 is 0 (and A23 is 0, as per section 2.3.4.1), then memory accesses are made using the protocol that the M68010 uses for accessing RAM. If A19 is set to 1, then memory accesses are made using the M6800 protocol. This protocol allows the local bus to interface with all the M6800 style peripheral chips on the M6800 bus. On the existing CPU board, only the bottom 9 bits (ie 512 bytes) are used to address individual peripherals.

So from the CPU's point of view, the M6800 peripherals appear in the memory map as a 512KB block at 4MB, and again at 5MB, 6MB, and 7MB.

2.3.4.4 Local Bus Peripherals

The key switch register appears as read only byte register on the M6800 bus at any of the addresses between 0x30-0x3F with A0 set to 1[2]. The lower four bits of this register represent the key position.

[1] i.e. the Fred Bus
[2] i.e. 0x31, 0x33, 0x35, etc.

The M6850 ACIA (the RS-232 controller) appears as two single byte registers at 0xC1 (which acts as the command and status register) and 0xC3 (which acts as the data register) on the M6800 bus. The 6850 interrupt line is also connected up to the M68010 as described in section 2.3.2.

The M6840 programmable timer appears as 8 single byte registers on the M6800 bus at the even addresses from 0x100 to 0x10C inclusive. It also has an interrupt line connected to the local bus as described in setion 2.3.2.

All the remainder of the M6800 bus addresses are either unused, or have undefined (and thus unsafe) effects when accessed.

2.3.4.5 The ELAN Station

Slightly surprisingly, rather than being being implemented as a set of Fred Bus registers, the interface to the ELAN Station is provided as a set of 4 registers accessed as 3 memory locations on the Local Bus. A ribbon cable is used to connect the CPU board to the ELANA board over what is called the *Front Bus*. This interface will make more sense when we examine the Front Bus, and history of the ELAN board in section 3.4.

The registers appear in the M68000 part of the address space as 4 bytes just below 512KB (ie 0x7FFFC-0x7FFFF), they are also duplicated at 0x17FFFC, 0x27FFFC, and 0x37FFFC.

The Status Register appears at 0x7FFFC, the Data Register appears at 0x7FFFD, and the Control Register appears at 0x7FFFF.

The ELAN Station is also connected to interrupt line 4 on the local bus. Asserting this line causes a level 4 interrupt to be raised on the CPU. The board can use this to inform the CPU of incoming words on the station/host interface.

The behaviour of the ELAN Station is one of the most complex parts of the Fred Machine and is described in its own chapter.

2.4 The Front Bus

The Front Bus is the second major bus that appears on most of the Fred Machine systems. No documentation was found on it, and it isn't mentioned at all in the APM

working documents. It is used to connect devices such as the 6809 kit board, the ELAN board, and the DMACK serial controller board directly to the Local Bus on the CPU board.

Physically it appears as another 96-pin socket on the front of each of the cards using it. A single ribbon cable with multiple connectors on it is the backbone of this bus, and each of the cards is plugged in to it.

Since no hardware documentation is provided, we must guess at its behaviour from the software that uses it. From the software, we discover that all the devices appear around 0x7F000. From this we can guess that 68010 CPU controls the bus, and the devices on the bus respond to the 68000's protocol for accessing memory.

Further examination of the software shows that some devices can assert interrupts on the CPU, so presumably the interrupt lines 4-7 on the local bus are also connected up to lines on the Front Bus.

2.5 The Memory Board

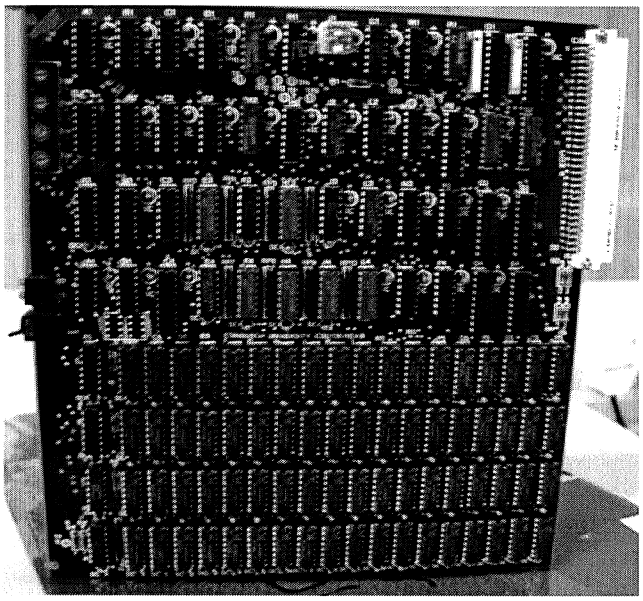

Figure 2.4: The 512KB memory board

The standard memory board for the Fred Machine is called as the CSD136. It holds

512KB of parity checked DRAM implemented using 64KB chips. The average access time is 230ns, which allows for approximately 4 million memory accesses per second. Given that the the CPU runs at 10MHz, and the 68010 takes several (sometimes many) clock cycles to execute each instruction, this is sufficiently fast for memory not to cause a huge performance bottleneck.

Several memory boards can be fitted in the one machine, and to distinguish between them, each board has 4 hex switches that are used to define the top 16 bits of the board's address in the Fred Bus address space. Since 512KB requires 19 bits of address space, the bottom 3 bits of the board's address switches must be 0. The APM working documents state that if the least significant switch is not set to 0 or 8, the board is disabled.

For debugging purposes, the board has 2 LEDs–an access indicator and a parity fault indicator. There is also a three way switch which controls the parity checking behaviour. The switch in the up position disables the parity checking. The switch in the middle position enables the parity checking, but disables the parity fault indicator. The switch in the down position has parity checking enabled, and sets the parity checking indicator to stay on once a fault occurs. The third state is the recommended state for normal usage.

Later versions of the memory board use 256KB chips rather than 64KB chips, and so hold 2MB of memory. Documentation was not avaliable for this board, but visual inspection of a 2MB board in one of the remaining Fred Machines show that it has 3 hex switches rather than 4, and presumably it must restrict the bottom bit of the least significant address switch to be 0.

2.6 The Level 1 Graphics Board

The level one graphics board provides a graphics controller with a write-only 1024×1024 pixel framebuffer to the Fred Machine. Two versions were made avaliable: a single board version capable of storing 4 bits of colour information per pixel and generating RGB TTL video outputs; and a two board version, storing 8 bits per pixel, with a 256 entry 16 bit colour map, and an analogue video output.

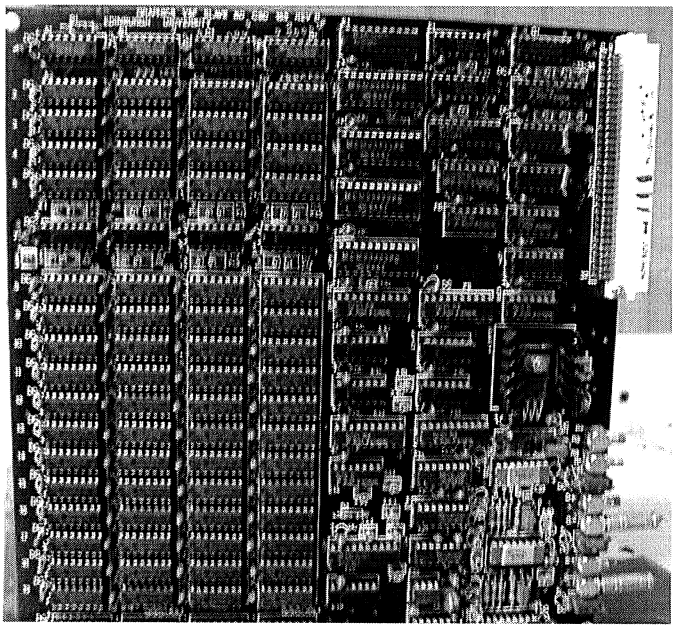

Figure 2.5: The Level 1½ board

Figure 2.6: The Level 1½ board's two layers

2.6.1 The Hardware

Physically, the graphics board has four address selection hex switches, similar to those on the memory board. The graphics board exposes 256KB of address space to the Fred Bus, so in a fashion similar to the memory board, the bottom 2 bits of the least significant hex switch must be 0.

The graphics board has an indicator LED which is lit if the framebuffer is being updated.

The graphics board generates a video signal suitable for displaying on a Mitsubishi C3419E monitor. This has a display resolution of 688×512 pixels, and since this is

smaller than the resolution of the framebuffer, a *window* of pixels from the framebuffer is displayed. The offset of this window from the origin is controlled by one of the graphics board's registers (the Display Register). Note that this window can wrap around from right to left (if the x offset is > 336), and from bottom to top (if the y offset < 512).

The layout of the Display Register allows us to do vertical hardware scrolling (as would be used in a terminal emulator), in 1 pixel increments. Horizontal scrolling can only be done in 16 pixel jumps. It is fortunate that the horizontal scroll is done in multiples of 8 pixels, as this simplifies the display hardware.

2.6.2 Display, and the Framebuffer

The coordinate system of the graphics board, in common with PostScript, graphics on the BBC Micro, and Cartesian geometry; and at odds with the likes of Apple's QuickDraw, Microsoft's GDI and the X Windowing System, places the origin at the bottom left of the framebuffer. Increasing x coordinates move from left to right, and increasing y coordinates move from bottom to top. Thus Level 1 graphics coordinates are in the range (0,0)-(1023,1023).

2.6.2.1 Bitmap Storage Models

Most modern graphics systems that expose framebuffers to a CPU treat the frame buffer memory as a single linear sequence of pixel values. For example, a system with 32 bit colour (8 bits of red, green, blue and alpha components) treats bytes 0-3 as being from the first pixel, followed by bytes 4-7 from the second pixel, and so on. Sometimes this is referred to as *chunky graphics*.

As an alternative to this approach, some systems use a different method: Memory is divided up into a set of single bit *planes*. In our example the 32 bits for the first pixel would be stored as 1 bit at the start of 32 different bytes of memory.

Both systems have their strengths and weaknesses. The chunky approach is good if photographic images or other bitmaps with many different colours are being dealt with, or lots of updates to non-sequential pixel locations are being performed, since only one operation needs to be performed to display a pixel (namely writing a single word to

the framebuffer). It performs poorly for dealing with situations where only a few (or perhaps just 2) colours are required–such as a terminal window. As the number of bits per pixel increases, the amount of data that needs to be written to the framebuffer increases in direct proportion. This means that a monochrome terminal window on a 32bpp framebuffer will require 32 times more memory traffic than a 1bpp framebuffer, despite the fact that the same image may be displayed.

In a planar system, this is avoided. If one wishes to display only a single pair of colours, as in a terminal window, one only needs to write to the appropriate plane of the framebuffer. However, a simple planar system is very weak at updating random pixel locations, as in a 32bpp system, 32 single-byte reads need to be performed, followed by 64 bitwise logical operations by the CPU, and 32 single-byte writes.

2.6.2.2 The Level 1 Framebuffer

The Level 1 graphics board provides a planar approach, with a little added hardware acceleration.

The board provides access to a single plane of 1024×1024 pixels, that appears as 128x1024 writable byte registers. Pixels are addressed from left to right and bottom to top. Within a byte, the most significant bit represents the leftmost pixel, and the least significant bit the rightmost. 8, 16 and 32 bit writes may be performed.

Writes to this pixel plane get mapped to the framebuffer as follows:

There are a pair of registers called the Plane Enable Register, and the Colour Register. The Plane Enable Register is a single byte, and effectively acts as a bit mask for pixel updates. If a bit in the PER is 0, then pixel updates to the given plane are disabled.

The Colour Register is also a single byte register, and the value written to that determines which colour planes will be set and cleared by writing to the pixel plane.

When a write is done to the pixel plane, each bit in the word written is examined. If it is 1, then the 4 (or 8) corresponding bits in the colour planes for the appropriate pixel are updated to the value stored in the Colour Register (as long as the appropriate PER mask bits are set).

2.6.3 Video Output

From the display point of view, pixel values are mapped to colours as follows:

On the 4 bit board, planes 0, 1 and 2 map to the red, green and blue guns being on or off. Thus 8 colours are displayable–black, red, green, yellow, blue, cyan, magenta, and white.

Plane 3 acts as a invert, or *cursor* plane. That is to say, it inverts the sense of the 3 other bits. This allows an efficient flashing cursor to be displayed purely in software without an unreasonable CPU overhead, and without requiring a readable framebuffer.

On the 8 bit boards, we must discuss another set of registers, namely the colour map. The colour map is a 256 entry table of 16 bit words. It appears as a writable set of registers on the Fred Bus.

In each colour map entry, bit 15 is a *flash* bit, bits 10-14 specify the blue intensity, 5-9 specify the green intensity, and 0-4 the red intensity. This means that each colour component has 32 possible values, leading to 32768 possible colours. When the flash bit is set, the display hardware alternates between showing black, and the specified colour every 16 frames.

When displaying pixels using the 8 bit boards, for each pixel, the 8 planes are used to give an index into the colour table. The values from the colour table are used to determine the intensity of the red, green and blue electron guns.

2.6.4 The Fred Bus Interface

As mentioned in section 2.6.1, the Level 1 graphics card exposes 256KB of address space to the Fred Bus. All of the graphics registers are write-only. The addresses are interpreted as follows:

2.6.4.1 Framebuffer

If A17 is clear (ie, the write is to the bottom 128KB), then we write directly to the pixel plane, with the effects described in section 2.6.2.2.

2.6.4.2 Control Registers

If A17 is set, then we write to one of the control registers.

When A16 is set (ie the address is between 192KB and 256KB), and we are using the 8 bit graphics boards, then we write to one of the colour map entries. The colour map entries are laid out as 256 32 bit words, with the color map being stored in the bottom 16 bits of each word. So, the bottom 10 bits of the address determine which entry is written to. If the bottom two bits are 0, or 1, the write is ignored. If it is 2, the most significant byte of the word is written to, if it is 3, the least significant byte is written to.

When A16 is clear (ie the address is between 128KB and 192KB) we write to one of the control registers. The control register is selected using the bottom 2 bits of the address.

When the address is 0, we update the Plane Enable Register from chapter 2.6.2.2.

When the address is 1, we update the Colour Register from chapter 2.6.2.2.

When the address is 2 or 3, we update the high and low bytes (respectively), of the Display Register. The Display Register contains the x and y offsets described in chapter 2.6.1. The top 10 bits contain the y offset (i.e. the topmost y coordinate), and the bottom 6 bits contain the x offset (i.e. the leftmost x coordinate) divided by 16.

This description of the Display Register is at odds with the in APM working documents. They describe the y offset as being the *Y Finishing Index*, which is the bottommost y coordinate according to the illustration in the document. However if we use the description from the working documents, some of the Fred Machine software does not display correctly.

Note that, in common with much of the Fred Machine hardware, the addresses for accessing the control registers are not fully decoded–the values of some of the address lines are ignored. This means there is no unique address for the 3 registers (and the colour map), and they appear many times within the address space.

Chapter 3

The ELAN Network

> To pretend, I actually do the thing: I have therefore only pretended to pretend
>
> Jacques Derrida, *The Post Card*

The ELAN network was a digital computer network set up in the University of Edinburgh in 1980. It was based on the design work that came from a pair of experimental systems, which were inspired by the Ethernet system developed at Xerox PARC[8], and the subsequent standardisation effort.

The design goals were slightly different to those at Xerox PARC. The Xerox PARC system was intended to connect a local network of graphical workstations (Xerox Altos[9] with local discs in this case) to each other, for email and light file sharing, as well as to a shared laser printer.

In Edinburgh, the system was intended to connect a large number of discless workstations to a small number of fileservers. Furthermore, the system needed to support a number of different architectures, including the VAX, the Fred Machine, and some Interdata minicomputers. Some of these systems, particularly the Interdata machines, had relatively little memory and processing power, so this motivated handling as much of the network stack as possible in firmware.

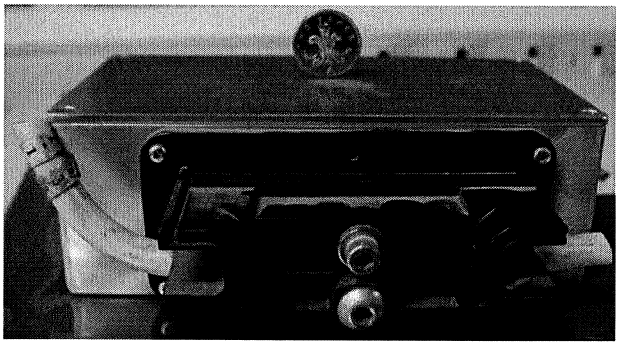

Figure 3.1: The ELAN vampire tap

3.1 The Physical Network

The physical ELAN network, and the physical, and link level protocols used on it are broadly similar to those of the standard 10Mb/s Ethernet.

The major differences are, broadly, as follows:

- Standard Ethernet runs at 10Mb/s, whereas ELAN's network runs at 2.1Mb/s.
- Standard Ethernet is designed for networks with cable lengths of up to 2.5km, and so the minimum frame size was 64 bytes (46 bytes of payload), to allow collision detection to work reliably. ELAN is designed with slower hardware, and smaller networks in mind, so the minimum frame size is 14 bytes (no payload). The maximum frame size in Ethernet is 1518 bytes, and in ELAN is 548 bytes.
- Standard Ethernet uses 6 byte MAC addresses to uniquely identify an Ethernet card. 3 bytes are a *Vendor ID* (allocated from a central registry), and 3 bytes are a *Vendor Serial Number*. ELAN's addresses were also 6 bytes–a 1 byte local Host ID, a 1 byte port number, and a 4 byte *Network ID* (allocated from a central registry). Note that this introduces elements of a higher level protocol into what is just a link level protocol on standard Ethernet.
- Standard Ethernet has a 16 bit *Service Access Point*, or protocol type field in each frame, indicating which protocol is carried in the payload, and a 32 bit CRC. ELAN also has the type field, but uses it to hold the packet type, whether

it is an acknowledgement, and a sequence number, again introducing elements of a higher level protocol. ELAN's checksum is a 16 bit CRC.

- The backoff algorithm, in case of collision, and the frame preamble bits are different between Ethernet and ELAN. The interpacket spacing is 3μs in ELAN, on Ethernet it is 9.6μs.

3.2 The Ethernet Station Hardware

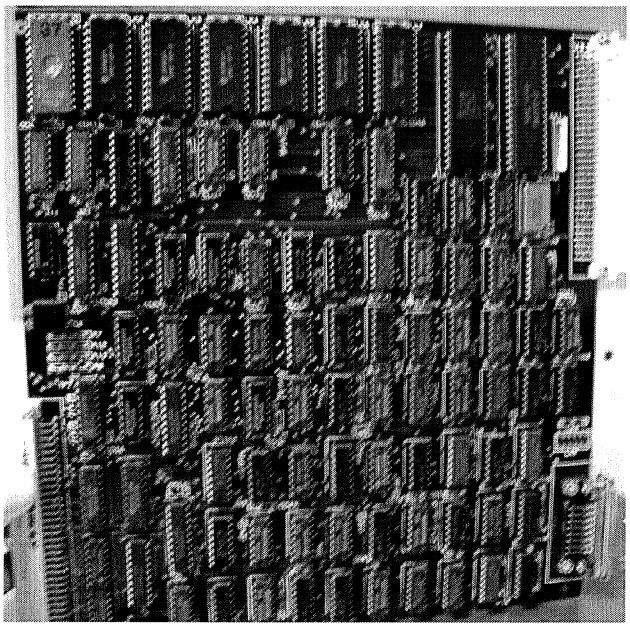

Figure 3.2: The ELAN Station

The ELAN Station boards consists of an Ethernet transceiver for sending and receiving Ethernet frames; a controller for buffering packets and handling acknowledgements, retransmissions etc.; and a simple interface to the host computer.

The transceiver has a number of features that save the host CPU from doing work.

- The receiving hardware has an optional filter that automatically drops frames that are not addressed to receiving station. It also has a 256 bit broadcast mask

which can be set to ignore broadcast frames to any of the 256 broadcast channels. On powerup all 256 bits are set to 0, ie all channels are ignored.

- The receiver has a 64 byte FIFO, and the transmitter has a 16 byte FIFO, allowing the controller not to have to poll all the time.
- CRCs are generated, and checked automatically by hardware.

The controller is a Zilog Z80 running at 4MHz. The firmware for the controller is stored in a 4KB ROM, and 4KB of RAM is used as both working memory and a buffer for 5 ELAN frames. The controller is connected to a DMA engine that has 4 unidirectional channels, both to and from both the host and the transceiver. The controller implements a simple connection oriented protocol with retries, acknowledgements, and sequence numbers on top of the raw ELAN frames. In ways it is reminiscent of a simplified version of TCP/IP, although unlike TCP/IP it would behave extremely poorly outside a LAN environment, and has no notion of routing frames between networks.

3.3 Services Provided

From the developer's point of view, the ELAN Station differs from standard Ethernet in one major way: Whereas on Ethernet, arbitrary frames are sent from a specified source to a specified destination, ELAN provides the notion of a port, which is roughly akin to a bidirectional UDP datagram socket in TCP/IP. The standard ELAN firmware provided 16 ports. Later versions of the firmware allowed up to 256 ports, although this change required a change in the host/station communications protocol. This later protocol is the one used in the emulation.

3.3.1 Ports

To explain this, firstly lets take a look at ELAN addressing. As mentioned earlier, ELAN MAC addresses are 6 bytes, as in Ethernet.

- Byte 0 specifies a host identifier. Valid addresses include 1-127, which represent a unique host, and 0, which represents a broadcast address. Bit 7 is reserved, and

is always set to 0.

- Byte 1 specifies a context. For unicast packets, this is a value between 0-15, and is the port number on the destination host that the frame is intended for. For broadcast packets, the value represents the broadcast channel number (0-255).

- Bytes 2-5, the last four bytes, specify a network identifier–a unique ID handed out by a central registry that specifies what LAN a station is on. On the ELAN network, this was set to all zeros.

The ELAN Station maintains a context for each of the ports. With the exception of port 0, a port must be opened, and associated with with a destination station (either unicast or broadcast), before frames can be received or transmitted on that port. An attempt to open an already open port implicitly closes the port first, before reopening it. After use, a port can be closed, and an attempt to close an already closed port silently succeeds.

An attempt to send a frame to a closed port, or the receipt of a frame from a source other than the one specified by the port context both fail.

Port 0 is regarded as special. It is the only port that can receive broadcast frames, and it is always open. Whereas with all the other ports, once a port is open, the host just sends raw data packets and the controller automatically appends the necessary addressing information, with port 0, there is no addressing context, so the host must specify a destination address with every frame. The host also receives a source address with every frame received from port 0. This allows port 0 to send frames to and receive frames from arbitrary stations without any special setup. In many ways, port 0's service is most like that of standard Ethernet.

One side effect of this is that initiating a transaction between two stations is slightly awkward: If a client station A wishes to connect to a server station B on some port, it must first send a frame on port 0, requesting that station B should open the appropriate port, and should then wait for incoming frames. After this, the client can attempt to connect to the desired port. This is quite unlike UDP/IP, where on the server side, the `bind` function can be set to allow an incoming socket connection from any host. The server can then can `listen` for connections from anywhere, and `accept` them.

3.3.2 Acknowledgement and Retransmission

Another less obvious aspect where ELAN differs from standard Ethernet is in its handling of acknowledgement and retransmission. Ethernet does not attempt to provide a reliable service, but ELAN does. In Ethernet, a frame may be corrupted (and thus dropped) with a probability of 10^{-5} or less, frame collision may occur, or the destination host may not be available. An Ethernet interface just sends frames out onto the ether, and as long as it doesn't detect a collision while it is transmitting, it doesn't attempt to retransmit. Whether the frame is actually received by another interface or not, is somebody else's problem, i.e. any further data delivery guarantees must be provided by higher layers of software.

In the Ethernet and ELAN frame format, there is a two byte field called the Service Access Point, or protocol type field. In Ethernet it is used to indicate the ID of what protocol stack should be used to process the payload–e.g. 0x0800 is IPv4, 0x6003 is DecNet Phase IV, and 0x809B is AppleTalk. In ELAN the type field has two purposes: the 7 least significant bits in first byte is used to identify the payload type, as in Ethernet. This is protocol 0 for filestore traffic. The most significant bit is used to indicate whether the frame is a network acknowledgement frame (1) or not (0). The second byte is a sequence number. When a port is opened, the first frame transmitted has a sequence number of 0. Each subsequent packet transmitted increases the sequence number by one. After the sequence number reaches 255, the next sequence number is 1.

When a host requests to an ELAN Station to transmit a payload, the ELAN Station wraps it up in a frame, buffers it and attempts to send it. The sending port is then blocked until an acknowledgement on the correct port and with the correct sequence number is received. If an acknowledgement is not received within a 35ms, it assumes that the frame was lost and retransmits it with the same sequence number. This repeats up to 30 times, with an extra 35ms being added to the delay each time, after which the station gives up on the packet and sends a negative acknowledge to the host. On the other hand, if an acknowledge is received in time, then the station can inform the host that the data was successfully send.

Note that while a given port is waiting for an acknowledge, anther port can be

sending. This ability to use ports in parallel partly makes up for the fact that this send/acknowledge protocol has no windowing[1], and as such is quite slow.

Note also that since a broadcast frame is not sent to any particular host, there is no acknowledgement for broadcast frames, and as soon as the frame is buffered, the station informs the host that it was successfully delivered.

3.4 The Host/Station Interface

There seem to have been some changes in the interface between the host and the ELAN Station over time. The 1982 ELAN document[3], describes the interface to appear logically as a pair of nine bit wide unidirectional channels, which can be used for transmitting or receiving data or control bytes. On these channels, control bytes were distinguished from data by having the topmost bit set to one.

However, when we examine the ELAN code used in all of the Fred Machine software, it refers to 3 memory locations as in chapter 2.3.4.5. These provide 4 single byte wide registers: a Status Register, a Command Register, a Data Register, and a Control Register. These registers are used to manipulate the 2 logical 9-bit host-to-station, and station-to-host FIFOs which are described in the ELAN document.

Delving further, it appears that the format of the command characters has evolved a little bit also. Rather than being a single command byte split up into a 4 bit command field, and a 4 bit data field, a single byte writes to the Control Register appear as the command. The controller then waits for writes to the Data Register, and treats them as the arguments to the command. Amongst other things, this means that the host/station protocol can handle having up to 256 ports, as the OPN and CLS (port open and port close) commands take a single byte argument.

To support extra useful functionality, the behaviour of the 4 registers is a little more complex than one would first suspect. The behaviour is described below.

- Reading from 0x7FFFC reads from the Status Register, and works much as one would expect–it gives the host information about the state of the ELAN Station.

[1] i.e. data transmission proceeds in a sequential send, ACK, send, ACK order, and thus speed is limited by the latency of the network. For a discussion of windowing see the RFC1323[12]

If there is no protocol data to be sent to the host, it returns a 0 byte, however if there is data in the station-to-host FIFO bit 0 is set to 1. If the next byte in this FIFO is a control byte then bit 1 is set. This is the same as the top bit being set in the protocol described in the ELAN document. If the next byte to be sent to the host is either data or an argument following an incoming control byte, then bit 2 is set. Finally, if interrupts are switched on, and the Status Register is non-zero, then bit 7 is set.

- Writing to 0x7FFFC is also possible, and behaves as writing to the Command Register. The Command Register controls the overall state of the card. Writing a 0 tells the Station not to assert interrupts on the CPU. Writing 6 to this register instructs the Station to assert an interrupt when it wants to inform the host of data on the station-host FIFO. When exactly it does this is not entirely clear, but it appears to be when a new word is added to station-to-host FIFO. The Command Register may support more commands than this, but these are the only two used by the Fred Machine system software.

- Reading 0x7FFFD, or the Data Register, does one of two things: If the first word on the logical host to station fifo is a data byte, or a command argument byte, then that byte is removed from the FIFO, and its value is returned to the host. If the first word on the FIFO is a control word, then the value of the word is returned to host, but the word is not removed from the FIFO.

- If a previous control command has been sent to the station that requires either a stream of data or an control argument to be sent to the station, then writing to the Data Register sends the information to the station one byte at a time. If the Station is not expecting information, then the byte is discarded.

- Writing to 0x7FFFF, the Control Register, does the equivalent of sending a word with the high bit set to the station in the protocol described in the ELAN document. That is, it sends a command character to the station. Depending on the command, this may require further argument bytes to be sent to the Data Register, followed by the results later being read from the Data Register. The individual commands are described later.

- Reading from the Control Register only works if the first word in the host to station FIFO is a control word. If it is, then the word is removed from the FIFO and returned to the host.

3.5 The Host/Station Protocol

The Host/Station protocol consists of control words (which can be sent from host to station, or station to host), and corresponding arguments, and data that goes along with them. Logically these words are 9 bits long, with the top bit signalling that the word is in fact a control word. A brief description of the control words, and how they were intended to be used follows:

- 0x04 – ENA – *Enquire Network Address*

  This control character is sent from the host to the station. The station responds with a control character SNA (Set Network Address), and a 4 data bytes containing the network address in the order they would appear in bytes 2-5 of a MAC address.

- 0x05 – ESA – *Enquire Station Address*

  This control character is sent from the host to the station. The station responds with a control character SSA (Set Network Address), and a single data byte with the host address.

- 0x06 – SNA – *Set Network Address*

  This control character can be sent from the host to the station. The station waits for 4 more data bytes, and then uses this as the network address of the station. The station can also send this control character to the host in response to ENA

- 0x07 – SSA – *Set Station Address*

  This control character can be sent from the host to the station. The station waits for 1 more data byte, and uses the data byte as the host address of the station. The station can also send this control character to the host in response to ESA.

- 0x08 – PTR – *Set Peek/Poke Pointer*

  This control character is sent from the host to the station. The station waits for 2 more data bytes (least significant byte first), and uses this as an address for future GET and PUT commands.

- 0x09 – BON – *Broadcast On*

  This control character is sent from the host to the station. The station waits for 1 more data byte. The byte received is interpreted as a broadcast channel number, and switches on the corresponding bit in the broadcast receive bit mask. The station will not discard future broadcast frames received with this channel number.

- 0x0A – BOF – *(All) Broadcasts Off*

  This control character is sent from the host to the station. It has no argument. The station immediately clears the entire broadcast receive mask – i.e. it will discard any future broadcast frames received.

- 0x0B – ETX – *End Transmission (of data)*

  This control character can be send by either the host or the station. It is used as the terminal word when transmitting a packet of data initiated by either DTX or OPN. It has no associated argument, as the port number can be implied from the rest of the data communicated.

- 0x0D – GET – *Peek byte*

  This control character can sent from the host to the station. The station immediately returns a PUT control character, followed by the value of the byte in the station's local memory from the address specified by the most recent PTR command.

- 0x0E – PUT – *Poke byte*

  This control character can sent from the host to the station. The station waits for 1 more data byte. That byte is written to the address specified by the most

recent PTR command in the station's local memory. The station can also send this control character to the host in response to GET.

- 0x0F – RESET – *Reset Station*

  This control character is sent from the host to the station. It effectively reboots the station, and returns it to its state on power on.

- 0x10 – RDY – *Ready*

  This control character can be send by either the host or the station. It is used as an affirmative response to a DTX or OPN control character from the other side. It is followed by a single byte indicating the number of the port that has just been opened, or has is ready to receive a data packet.

- 0x20 – STX – *Start Transmission (of data)*

  This control character can be sent by either the host or the station. It is used at the initial word when transmitting a packet of data initiated by either DTX or OPN. It is followed by a single byte indicating which port the frame is to be sent on.

- 0x30 – DTX – *Transmit Data*

  This control character can be sent by either the host or the station. It is used to indicate that one side wishes to send an ELAN frame to the other. This occurs either when the host wishes to transmit data on the network, or when the station has received a packet that should be passed on to the host. The control character is followed by a single byte indicating the port number associated with the frame. The other side responds with a RDY. Once this is received, sends an STX, followed by the data one byte at a time, then a final ETX. Note that the entire operation from DTX to ETX is regarded as atomic. It is a protocol violation for either side to attempt to start other transmissions, or embed other control characters inside one of these operations.

  Note that DTXs on port 0 must prepend the user data with the 6 bytes of the MAC address (transmitted in order from byte 0 to byte 5). The user data must be no more than 532 bytes long.

- 0x50 – NAK – *Negative Acknowledge*

  When the station has attempted to send and resend a frame to a remote host without success, and finally times out, the station releases the buffer, and sends NAK control character to the host, followed by a single byte representing the port number the packet was transmitted on.

- 0x80 – OPN – *Open Port*

  This control character is sent from the host to the station. The station waits for the host to transmit one more byte. This byte is the port number that host wishes to open. The station replies with a RDY. The host then sends an STX, followed by 6 bytes containing the MAC address that should be associated with the port (transmitted in order from byte 0 to byte 5).

- 0x90 – CLS – *Close Port*

  This control character is sent from the host to the station. The station waits for the host to transmit one more byte. This byte is the port number that the host wishes to close. The station closes the port, and doesn't have any response.

- 0xA0 – ERR – *Error*

  This control character is sent from the station to host. It is sent in response to a command that doesn't make sense, or is invalid – such as trying to OPN an open port, or trying to send a RDY, or STX in the wrong context. If the control character that prompts this error had a port number, the error character is followed by a single byte representing the port number.

- 0xC0 – ACK – *Acknowledge*

  When the station receives an acknowledgement from a remote station for a frame that was transmitted to it on a particular port, it sends an ACK control character to the host followed by a single byte representing the port number the packet was transmitted on.

In addition to the above commands, the ELAN documentation describes a *protocol-less* mode that allows the user to send an arbitrarily long stream of data from station

to station. The splitting up of data into individual packets, and their transmission are handled by the station itself. The user simply sends an SOT control character, followed by a port number byte, and a stream of data terminated by any control character. This mode was intended to make it easy to write minimal bootstrap code for simple host processors. The Fred Machine software does not take advantage of this mode, the Fred Machine filestore does not use it, and indeed none of the available firmwares for the ELAN Station implement it. The only reference to it's use is on the Interdata based filestore when bootstrapping discless Interdata workstations.

Note that there was a certain amount of difficulty in pinning down some of the ambiguities in the behaviour of the host/station protocol. The ELAN documentation gives example traces of how the protocol should be used, but doesn't define all the different failure modes, and certainly doesn't provide a state transition diagram. To attempt to fill some of the gaps, we examined the ELAN firmware, the Fred Machine system software and Christopher Lisle's ELAN emulation. Hopefully these sources combined should give a reasonably accurate picture of how the protocols worked.

Chapter 4

The Fred Machine System Software

> When you see a good man, try to emulate his example, and when you see a bad man, search yourself for his faults.
>
> Confucious, *The Analects*

As stated in the introduction, examining and using the software on the Fred Machine is the prime reason for emulating the system. To examine the software in general, we will start with the system software since everything else relies on the services provided by it.

From a modern point of view, and from the point of view of somebody who is used to a roughly UNIX-like operating system design, the Fred Machine system software seems rather odd. Some of its features seem quite interesting and sophisticated, and in other areas (and with the benefits of hindsight) it seems quite lacking.

In many respects, it becomes quite clear that the system software is very much a product of its history. It started out as a minimal I/O library and runtime system for the IMP compiler, and grew into something bigger. When one realises this, a great many things start to make sense.

A few observations about the system may be in order:

- It is not UNIX. There is no notion of a separation between the kernel and the userland. The kernel is not privileged above other software, and does not provide an abstraction of any hardware devices to user processes.

- It is really not UNIX. There is no multi-tasking, either pre-emptive, or co-operative–just one process at a time. There is a single, flat, unprotected address space, and the system software, libraries, memory mapped devices, and the currently running process are all equally accessible.

- It can only be used as a networked machine. Assumptions are made in many places that file I/O is only done by speaking the filestore protocol. There are no notions of *pluggable filesystems*, or a *vnode layers*. Even the file I/O code itself is scattered around a number of libraries[1]. There is a notion of a currently logged on user, but it is very much tied into being logged into the filestore.

- The services provided by the system 'kernel' are slightly ad-hoc–they include a small low level debugger, interrupt based serial I/O, simple terminal I/O, interrupt timer handling, interrupt handling for communications with the ELAN board, handling some of the filestore protocol, a package that provides access to 4 abstract I/O streams, string handling routines, and routines to manipulate dictionary and array structures.

4.1 Bootstrapping

In the Fred Machine, the first 4KB of memory consist of boot ROM. This ROM sets up some basic hardware, loads the system file off either hard disc, or a filestore, and then passes control to the system file. The procedure occurs as follows:

- When the Fred Machine is powered on, the M8010 automatically reads in the first 8 bytes of memory and uses the addresses from these to set the initial stack pointer, and program counter, and starts executing code. Since the boot ROM is located at the bottom of memory, these 8 bytes are read from ROM, and are used to point the CPU at the ROM's initialisation routine.

[1] One unusual instance of this is the CLI program, which generates raw filestore requests by hand to load executable data, rather than using stream I/O. Presumably this is done in the name of efficiency, as stream input is read one byte at a time.

- The boot code contains a vector table used by the 68010 to store the addresses of routines to call when various exceptional events occur (such as divide by zero, bus error, etc.). To make this flexible, the ROM contains stub routines that just jump to entries in a corresponding *pseudo-vector* table in local RAM. The system file will be loaded into this location, and this allows the system to change its interrupt handling at any time, without needing to change the boot ROM.

- The boot code initialises the serial controller, initialises the ELAN Station, and sets the mapping registers to do a *transparent* mapping[2].

- The boot code determines the amount of available memory by writing values to every 32 bit word of external memory until the Fred Bus detects an invalid write and causes a bus error. Since the memory must be laid out sequentially, when a bus error occurs, the boot code concludes that it has moved past the highest addressable word of memory.

- If a key is held down, the boot ROM prompts the user for a file and filestore to use as the system file, otherwise it uses a default (the file `FMAC:NSYS` from filestore 0). It then opens a connection to the filestore, reads in the system file into local memory (starting at 4KB) and transfers control to the system.

4.2 The System Software

As mentioned earlier, the system handles a miscellany of low level behaviour. When it boots, it initialises a number of housekeeping variables, including the lowest free memory location, and a number of systemwide dictionaries used for file, command and library management.

Here are some observations on various features provided by the system.

- In a manner similar to the *cooked mode* terminal access in UNIX, a number of control characters are handled specially by the terminal code. Ctrl-Y kills the current program, and returns to the command line, Ctrl-T drops into the built in

[2] A transparent mapping is one where the translated and untranslated addresses are identical.

debugger, Ctrl-P acts as an escape character allowing the next character to be entered verbatim; Ctrl-S and CTRL-Q enter and leave auto-freeze mode–a mode where the terminal displays one screen of data at a time and waits for a key press to continue, CTRL-X deletes all the characters on the current line.

- The system provides 4 input and 4 output streams. The stream routines handle the low level reading and writing of blocks of data from a file into a buffer as necessary, and the user is given an API that allows them to read and write using the `readsymbol`, `printsymbol` and `printstring` functions.

- As would be expected of a language runtime, the system handles serial I/O and gives other programs an abstraction of a terminal. The terminal is tied into the streams I/O system, so opening the file called `:T` gives the user access to the terminal as if it is a file. This is similar to `/dev/tty` on UNIX. Another special file is `:N` which plays the same role as `/dev/null` in UNIX and `NUL:` on VMS. This functionality is hardwired into the streams code on a very low level–there is a check in the `openinput` function for these two special files.

- The system keeps track of the number of milliseconds that the computer has been turned on, and configures the 6840 programmable timer to generate timer interrupts at 10Hz. The interrupt routine updates the millisecond counter, so it is accurate to 100ms.

- An interesting part of the IMP runtime is the ability to signal *events*. Events are IMP's implementation of exception handling. It is possible to register an event handler which is called whenever an exception is thrown (or an event is signalled, in IMP parlance). Since this is provided at a low level in the system, the system itself uses it as a part of its stream API, and as a means of notifying user code that various sorts of problems have occurred.

  One place where events are used is to signal various processor exceptions, such as divide by zero occurs, or attempting to execute an illegal instruction, or attempting to call a reserved TRAP. In this regard, events are used in a way not unlike UNIX signals.

A more interesting example is how events are used as part of the streams API. To read in an entire file, the programmer sets up an event handler to catch the end of file event (event 9,1), and then goes into an infinite loop calling `readsymbol` to read the file one byte at a time. When the end of the file is reached, the handler will be called automatically.

- There is a basic debugger as a part of the system. It can single step, provide memory dumps, the user can set breakpoints. Finally the user can set a *protected* 32-bit word. When the protected word is modified by an instruction, the debugger is entered automatically.

- The system makes heavy use of 4 data structures called dictionaries. Dictionaries are fixed sized pools of memory that allow strings to be associated with index strings (like in a real life dictionary). They are built in such a way to allow substring matching on the index string, and once a dictionary is full, a new dictionary can be chained onto the end of the existing one to allow further items to be added.

- The system is responsible for keeping track of a list of symbols exported by executable files. A program called `INSTALL` can be used to load all the symbols from an executable into the external dictionary. In the dictionary they are associated with the file they came from and a memory location if they are loaded into memory. They are then available for use by other programs. Programs can make use of external symbols, and when a program is being loaded, the loader checks that all the external symbols are available in the external dictionary. If they are available, it loads all the necessary executables into memory, and allows the calling program to run. This effectively gives the system a shared library facility.

- The system manages a command dictionary for the CLI. The command dictionary maps a command line entry onto a shorter equivalent string. The functionality is similar to that of aliases in the UNIX shell, or command verbs under VMS.

- The system manages a file dictionary for the CLI. This associates a start address with a given command symbol. The idea is that the user can allow certain frequently used commands to be loaded into memory permanently. The CLI can search the file dictionary to see if it needs to load an executable to run the command, or if it can just run the command directly from memory.

Once the system has finished initialisation, it loads the CLI executable from the file `FMAC:ACLI.MOB` using a file stream, updates the system so that it appears to be a normally loaded executable, and then transfers control to it. In ways, this could be regarded as being the same as the way that a UNIX kernel manually sets up the process structures for the first process on the system and loads the `init` executable into it.

4.3 The Command Line

The Command Line Interpreter on the Fred Machine is responsible for taking command line input from users, and using it both to maintain environment settings, and to load and run programs.

As far as environment goes, the CLI manages the free memory on the system, keeps track of which I/O streams are open, provides default terminal settings and default event handling routines.

There are two types of binaries on the Fred Machine–old-style binaries, and new style *FE02* binaries[3]. Old style binaries, are just a simple binary object format, with no references to anything other than common system routines. These can be run directly by the CLI–it just loads them into memory and jumps to their start address.

The new style binaries have import and export tables, as well as code and static data sections. This allows them to export functions and data, in other words they can behave as shared libraries. As either programs or shared libraries, they can also access functions and data from other binaries. To run the new style binaries, the CLI runs a program called `FMAC:MEXRUN`. `MEXRUN` is essentially the dynamic linker for the Fred Machine and is responsible for making sure that references in the binaries' import

[3] So called because the file header begins with the bytes 0xFE and 0x02

table are resolved, and that any necessary libraries are loaded into memory before the program starts.

The CLI and `MEXRUN` use and update the file and external dictionaries to store the runtime information required to do dynamic linking.

Three other utilities that are used for dynamic linking are `INSTALL`, `PRELOAD`, and `REMEMBER`. `INSTALL` reads an executable file, scans its export table for symbols, and adds these symbols (and the file they belong to) to the external dictionary. After this is done the executable is effectively registered with the system as a shared library.

`PRELOAD` compliments `INSTALL` by taking an executable, loading it into memory, and marking the executable so that it stays resident. This is useful for commonly used libraries (such as terminal handling libraries etc.), as the standard behaviour would be to load the library on demand each time a program that uses it is started, and to free up the memory once the program finishes. Using `PRELOAD` in these circumstances saves a lot of unnecessary network traffic.

`REMEMBER` behaves in a manner similar to `PRELOAD`. It marks the executable so that it stays resident, but it doesn't actually load it into memory. The next time a program does use the executable however, it is loaded into memory and stays there. So `REMEMBER` is a load-on-demand version of `PRELOAD`.

There are four kinds of commands that can be given to the command line:

- A string that begins with a '!' is ignored completely, and can be used as a comment.

- A string can be associated with a longer command name using the syntax `SYMBOL =COMMAND STRING`. When the CLI encounters this, it adds a new item to the command dictionary, and any future commands get checked against dictionary entries while being parsed

- A program can be run with the syntax `PROGRAM ARGUMENTS`. The first part of the input string gets checked against the command dictionary, and if there are matches, the input is translated appropriately. Finally the input is broken up into a *verb*[4] (or program name) and arguments, and the CLI attempts to run the

[4]This name probably was inspired by verbs in VMS's DCL.

program and pass it the appropriate arguments. The CLI automatically appends `.MOB` to a program name if it is not included in name already.

- A script or *obey file*[5] can be run with the syntax `@SCRIPT ARGUMENTS`. As with programs, dictionary matching and translation are performed. The CLI automatically appends `.COM` to an obey file name if it is not included in the name already. Running an obey file simply involves the CLI reading in the obey file one line at a time, and behaving as if the user had typed in the same commands on the terminal.

Unlike the command line shells on UNIX, there are no general programming constructs such as conditionals, loops and branches. This has this virtue of keeping the CLI small and simple, rather than turning it into a miniature programming language.

To extend the bootstrap process a little further, and to allow for easy user customisation, the CLI automatically runs the obey file `FMAC:ASTARTUP.COM` when it is started. This file typically contains a number of command assignments, and some `INSTALL` and `PRELOAD` commands to set up commonly used libraries. An example follows:

```
! Set up system
files=fmac:files
ie=fmac:ie
install=fmac:install
preload=fmac:preload

f=files
ml=ml:nufam -h 1200 ml:smlcore.exp

install l:pam,l:maths,l:terminal,level1:graph

preload fmac:lib.mob,l:pam.mob

! Use a VT220
```

[5]This term comes from the OS on the BBC Micro which had a similar notion.

```
terminal=vt220
install l:wtlib

@custom:setup.com
```

4.4 The Filestore

A filestore server, in its later incarnations, was a Fred Machine with one or more internal hard discs and an ELAN Station, that was connected to the ELAN network and served out files to client Fred Machines. Typically a filestore had between 100MB and 300MB of storage, and could serve up to 31 simultaneous clients. We will not discuss the file store in great depth, as it has already been described elsewhere[1][4], but an overview of the service provided and some of its semantics are instructive for understanding the Fred Machine software.

The filestore data structures that are stored on disc were vastly more limiting than the filestore protocol actually allowed for. For example, on an unfragmented disc, a directory could only store at most 102 files. We will not look at the implementation of the filestore itself, but shall instead examine the service that was made available to clients. We do this with a view to determining what features would have to be provided by filestore software that running in the host environment.

4.4.1 Directories, Names and Metadata

The filestore provides the client machine with access to an arbitrarily large list of directories, each containing an arbitrarily large list of files. In the most common filestore, directories could not contain other directories, so it is not a hierarchical system.

Each directory was owned by a single user called an owner, and all the files within that directory shared that ownership. Each owner has a unique name (which is the same as the directory they own), a password, and an administrator controlled disc quota.

All names in the filestore are case insensitive.

An owner-name consists of up to six alphanumeric characters, the first of which is a letter. The dollar sign is reserved as a special owner-name.

A filename consists of up to twelve characters. The first character must be a letter or a dollar sign, and the remainder can be a letter, a digit, or a period.

A full filename is specifies the owner-name and the filename by separating them with a colon e.g. `FMAC:SYS`.

Each file has several pieces of information, or metadata associated with it. One piece of metadata is the filename, another is the list of block extents defining where the file is stored on disc (this is not user visible), another is the date stamp, which stores the creation date of the file to the nearest minute. A final piece of metadata is the file's attribute. This consists of the file permissions (described in chapter 4.4.3), and an archive bit which can be set to Archive or Vulnerable. Files with Archive set are automatically archived if they have been modified since the last backup. Vulnerable files which haven't been used for a period of time are automatically deleted.

Unlike UNIX, a directory is not a file, and can not be read directly using file operations.

4.4.2 File Creation Semantics

The filestore has a interesting feature which is not seen in UNIX-like filesystems: Files can be either permanent or temporary. Permanent files stay on disk until they are explicitly deleted, as one would expect, and their disc usage is deducted from the owner's quota. Temporary files only exist for as long as the owner is logged in, and do not use any of the owner's quota. In one way, this gives the user a similar facility to the `/tmp` directory in UNIX, or `SYS$SCRATCH` on VMS. Temporary files are distinguished from permanent files by having a name starting with a dollar rather than a letter. Since temporary files only exist while the owner is logged on, they can only be created by the owner. Since the owner can be logged on multiple times, the filestore keeps a reference count of how many times the owner is logged on, and does not delete their temporary files until the last session logs out.

Another filestore feature which differs from the UNIX model is the notion of transient and subliminal files, and the semantics of file creation. Under UNIX, when a new file is created with the same name as an old file, the directory entry of the old file is removed, and when the the use-count of the file goes to 0 (i.e. all processes have closed

the old file), the file's storage is reclaimed. On the filestore, when a new file is created, it has the property of being a transient file. It gets new storage allocated to it, and when closed becomes a permanent file.

If for some reason, the client or filestore crashes before a transient file is closed, the file remains transient. The only operations that may be performed on this file are to delete it, in which case it is removed; or to rename it, in which case it becomes a permanent file.

If a new file is created with the same name as an existing one, two copies of the file exist–the transient one, and the permanent one. In this situation, attempts to open and read the file operate on the old permanent file, and any delete or rename operates on the transient file.

Subliminal files are files that are due to be deleted. A file becomes subliminal if it is being used by a client when another client attempts to delete it. A file created with no name is subliminal for all of its existence. Subliminal files are deleted when the last client using it closes it. In this sense, they are very similar to UNIX files that have been deleted–they are still backed by disk space, but they no longer have a name, and no new processes can access them.

4.4.3 File Security

Each file has security metadata associated with it, in much the same manner as UNIX's file mode bits. There four levels of permission, and they can be set for two kinds of user: the owner and the public. The four levels are

- Free – The file can be read from, and written to. The owner may also delete this file.
- Read – The file can be read from.
- Obey – The file can be read internally by the filesystem.
- None – The file may not be read from at all.

The permissions are constrained so that the owner must have at least the same level of access to a file as public users.

Chapter 5

The Emulation

If you haven't got it, fake it!

Victoria Beckham

We imitate; and what is imitation but the travelling of the mind?

Ralph Waldo Emerson, *Self-Reliance*

As was mentioned in the introduction, the main purpose of this project is to provide a sufficiently accurate emulation of the Fred Machine system to be able to explore a substantial amount of the software written for the environment. Whenever questions arose as to how precise the emulation should be, or how the emulation should interact with the host environment, decisions were made on the basis of supporting existing software. Verisimilitude, while desirable, is a secondary goal.

Since the programming work in this project builds on the work done by Christopher Lisle[1], we briefly examine the functionality of his emulation, before describing the enhancements made, and new features added.

5.1 The Existing Emulator

The existing Fred Machine emulation consists of three programs written in a mixture of C and C++ and designed to run in a UNIX environment with TCP/IPv4 available. On

examination, much of the difficult work in this centres around attempting to provide an accurate emulation of the ELAN networking from the point of view of the Fred Machine's CPU.

We discuss the networking in the next section, but in a nutshell, it consists of a nameserver process, which maps ELAN hosts onto IP addresses; and C++ filestore, which communicates with the emulator via ELAN frames encapsulated in IP datagrams.

The Fred Machine emulation itself centres around an open source 680x0 emulator called Musashi written by Karl Stenerud for the MAME project[10]. This provides a thorough emulation of almost all of behaviour of Motorola's 68000, 68010, and 68020. The 68451 MMU is not emulated, but this is unimportant, as the CPU board's MMUs are never used by the Fred Machine software. One feature of the CPU emulation that is missing is a means of handling bus exceptions when unavailable memory is accessed. This is handled in the existing emulator in one specific case–when the CPU writes a 32-bit word from a data register to memory. This is sufficient for the boot ROM's memory probe, but there are other programs that check for the presence of cards in a similar way, but with different instructions, and these do not work as expected.

The serial port emulation works by having a thread that repeatedly reads bytes into a buffer using `getchar()`, and provides these to the emulation (along with interrupts) as necessary. This scheme usually functions reasonably well, although it seems odd and a little 'un-UNIX-like' to use a thread for this purpose. However, there are some odd situations where the emulator seems to randomly refuse input from the console on boot up. Also, the user must remember to turn off UNIX's handling of control characters for signalling, and UNIX's *cooked mode* line buffering using the `stty` utility.

The 6840 programmable timer is not emulated at all, and no attempt is made to run the emulator at a speed close to that of the real hardware–instead it runs as fast as possible. This is unimportant for a basic emulation, but becomes an issue with interactive software, particularly for games. Less frivolously, it is also important for any software that uses the timer to estimate how much CPU time some code has used, or that attempts to provide timeouts for I/O code.

The emulation of RAM is hardcoded into the software as a single block of memory

with no details of the hardware included. The emulated machine has 1MB of RAM, and this can only be changed by recompiling the emulator. This is a little annoying, but is simple to fix. Memory accesses take no emulated time to perform, and none of the lower level details of the Fred Bus are emulated. The lack of cycle counting for Fred Bus transactions means that even if the CPU were constrained to run at 10MHz, the emulation would run faster than the real machine. Again, this situation could be catered for, but it is unclear whether it is particularly desirable or not. The lack of emulation of Fred Bus arbitration is probably the most sensible approach–the only situation where one would need to care about those details is when multiple CPU cards are in use, but since, apart from a research OS, no software was available for this environment, it seems unnecessary to emulate it.

The Fred Machine case has a 5-way key that appears as a read only register on the CPU Local Bus. This is not emulated, and there seems to be no pressing need to emulate it, as no software was found that uses it.

The CPU card has a reset switch on it, and no means is provided to emulate this, although it would be fairly simple to do this. One question that would arise in trying to do so is how to provide an unobtrusive user interface for it. A possible means would be to have the emulator react to a UNIX signal such as `SIGUSR1` by resetting the CPU.

The CPU emulation provides a basic debugger. This has some useful features, including breakpoints, single step, dumping data, address and mapping registers as well as memory dumps. These mostly duplicate the functionality of the Fred Machine's built in debugger, but the emulation also provides what could be a very useful feature: a disassembler. Unfortunately, in practice the whole debugger is next to useless due to the way it interacts with the console input. If the user single steps then everything works fine, but as soon as the user tries to run until a breakpoint all further console input gets read by the console input thread. Once the breakpoint is reached, the emulator returns to the debugger and awaits user input, but since input is now consumed by the input thread, this can never happen.

It might not seem so terrible to debug the boot process by single stepping, except for the fact that the boot ROM has a loop that tries to write to every 4 byte word of memory. On 1MB machine, this involves single stepping through about 500,000

instructions. Clearly, this is not practical. There doesn't appear to be any easy way of fixing this in the emulator either, short of completely rewriting the console handling code, or changing the debugger's user interface.

5.2 ELAN And The Emulated Filestore

The most complex component in the Fred Machine, from the emulation point of view, is the ELAN Station and the filestore it connects to. When emulating this, the previous project made a few design decisions that seem questionable.

5.2.1 ELAN

Since the ELAN 2.1Mb network never existed outside Edinburgh University, is no longer in use, and there are no ELAN Stations for modern computer hardware, sending ELAN packets over an ELAN network is completely out of the question.

Since ELAN is quite close to standard Ethernet, another approach might be to send raw ELAN frames directly over Ethernet. Unfortunately, due to the differences in frame format, this is would be quite problematic, and would cause many nasty interactions with host software. Firstly, the 16-bit protocol type field, used in modern Ethernet to route frames to the appropriate software to do protocol decoding is used very differently in ELAN. As an example, type 0x809B means AppleTalk in Ethernet, but would mean the acknowledgement for the frame with sequence number 155 in ELAN.

Secondly the MAC addresses under ELAN Ethernet, rather unfortunately put the host ID and channel ID as the first two bytes of the MAC address. In modern Ethernet, these bytes are the first two bytes of the 3 byte Vendor ID. The Vendor ID is allocated by a central registry, and cannot be changed freely. So the chances are good that many Host/Channel ID pairs would end up conflicting with already used MAC addresses.

Thirdly, modern Ethernet has a 32-bit CRC rather than a 16-bit CRC, and different restrictions on maximum and minimum frame sizes, which are not compatible with ELAN.

All of these problems could be worked around by encapsulating ELAN frames inside Ethernet frames, but then other problems arise such as the user requiring special

privileges to be able to read raw Ethernet frames, and the fact that these frames aren't routable across networks. In short, using Ethernet frames is too low level, and we should search for an alternative solution.

The method used by the previous project encapsulates ELAN frames in IP datagrams as follows.

There is a small nameserver program that maintains a mapping from each of the 127 possible ELAN Host IDs to a particular IPv4 address and port number. This communicates with the other software over a simple TCP/IP based protocol. The client connects to the nameserver, sends a single byte Host ID to the nameserver, and the nameserver replies with a null terminated string in the format `address:port`.

Software that wishes to communicate over emulated ELAN chooses a Host ID, retrieves the appropriate IP address and port number from the nameserver and sets up a thread that listens for incoming datagram connections on that port. Once it has done this, it can receive replies, and so is ready to send frames.

To send an ELAN frame, it retrieves the appropriate address/port from the nameserver, opens a datagram connection to that address, sends a raw ELAN frame, and closes the connection. The receiving thread will then get a matching reply from the remote host, and can deal with it as it sees fit.

This is functional, but has a couple of problems. Firstly, it is quite inefficient. For every single frame sent, a lookup has to be done, a connection opened, and a connection closed. Then, the receiver has to do all this again, just to send back an acknowledgement frame.

Secondly, by its very nature, it requires the programmer to use threads, even if they just want to implement a simple client that just sends a couple of frames. The encapsulation protocol requires the client to listen for incoming connections to receive responses from the remote host. The ELAN protocol requires the client to resend frames to the remote host if it doesn't receive an ACK, and also that a frame must be acknowledged, or dropped before another frame can be sent to the same destination. Because of these two facts, the client must, at the very least, implement a miniature 'server' that receives ACK frames. To listen for incoming connections using IP sockets, requires using the `listen` call, which is blocking, and so cannot be in the main

thread of execution. This forces the developer to use threads, or some other form of concurrency, and makes it especially difficult to debug problems.

Thirdly, the specific implementation of this encapsulation used in the previous project exposes many of the implementation details to the developer, and makes it difficult to do anything without having to use more threads, where it should not be necessary. Ideally, one would like to be able to write something like

```
elan_open_port(station, 1);
elan_send(station, 1, buf, 128);
elan_recv(station, 1, inbuf, &len);
elan_close_port(station, 1);
```

One feature that is completely absent from the ELAN emulation is support for broadcast frames. It is unknown how much they were used, but it would be a nice feature to support.

We investigate solutions to these problems in chapter 5.4

5.2.2 The Emulated Filestore

One approach that could have been used to emulate the filestore required by a client Fred Machine would have been to emulate one of the hard discs, and the corresponding controller used in a Fred Machine filestore. It would then be possible to take a raw dump of an actual disc from a filestore machine, and simply run a Fred Machine emulation to provide a filestore. This would work, but has some practical problems. The main one is how would one obtain a disc dump in the first place. The second problem is that this doesn't give the user any means of transferring files between the host environment and the emulated environment. This is a good example of trying to decide on what level to emulate a system. A low level emulation is more accurate, but difficult to set up, difficult to work with, and interfaces poorly with the host system. This is particularly true when both the host system and the emulated system are trying to solve a similar problem (in this case, storing files on a hard disc).

Another approach, which is the one used, is to provide a native[1] program that reads

[1] i.e. running directly on the host environment

normal files on a local hard disc, and also speaks the filestore protocol to Fred Machine clients over the simulated ELAN. This approach is made practical by the fact that the filestore protocol is reasonably abstract and is not tied to specific details of how the files are stored on disc. It is also quite well documented, although some details aren't very well specified. Fortunately the source code to some versions of the filestore were made available.

As has already been described earlier in this chapter, the filestore semantics for file creation are a little different from UNIX's semantics, and require special handling. The existing filestore implementation avoids this by providing read-only access to the filestore.

Considerable complexity arises in the filestore server because of the poor abstraction of the ELAN networking simulation. Each incoming request starts off a filestore transaction, and each of these must be handled by a new thread. The locking constraints, and interactions between these threads and the rest of the code are not particularly clear, and make it rather difficult to debug the intermittent problems that occur. On occasions the filestore does not respond to the client when it should, and on other occasions it is possible to crash the filestore.

The presence of a native filestore client to stress test the system would have been of considerable utility–it was frequently unclear in what parts of the code base a problem was occurring: was it in the filestore code, or the networking emulation, or the ELAN Station emulation, or in the emulated Fred Machine client code?

5.3 Level 1 Video

The existing Fred Machine emulation provided no video emulation, but it is mostly straightforward to implement. The Level 1, and Level 1½ video are well documented.

To make the graphics code simpler, and to increase the performance of the emulation, the emulated framebuffer is stored internally as *chunky graphics*–one byte per pixel, rather than the 8 separate bit-planes that are used in the real hardware. This doesn't cause any compatibility problems, as the framebuffer is write only. The set of pixels that represent the image displayed on the monitor is stored as a separate 16bpp

framebuffer.

One design decision was to try to find a portable way of displaying the graphics on the host machine. In the end, partly out of convenience, and partly for portability, it was decided to make the framebuffer visible by making it an RFB server. RFB is the Remote Framebuffer protocol, and was designed as part of a project at AT&T called VNC[11] to allow arbitrary machines to display their graphics over a TCP/IP network. It also allows remote machines to control the keyboard and mouse of these machines. Viewers and servers have been written for many different platforms, from the Palm Pilot, and Apple Newton, to Windows, and X. This project uses a library called libvncserver to handle the low level details.

This decision does have the downsides that the user must run a separate program to see the Fred Machine's video, and that the network traffic and context switches generated make it slower than a direct solution. However, none of the rest of the graphics emulation hinges on this decision, and ultimately a different display solution could be used, such as Cocoa under MacOS X, or a shared memory pixmap on the X Windowing System.

Initially the graphics emulation was quite naïve. Screen refreshes were performed directly in response to writes to the video board. Some writes to the video board involve 8 pixels being updated, and this might involve corresponding pixels on the screen (i.e. the RFB framebuffer) being redrawn. Other writes involved updating the entire screen (such as when the colour lookup table was modified, or when the offset register was updated). These required all 688×512 pixels on the screen to be scanned and updated instantaneously, thus making it considerably slower than the real hardware. One place where this caused serious problems was in emulated code that updated all 256 CLUT entries in a loop. On real hardware, this should take milliseconds, however the emulation would take many seconds, and 255 extraneous screen refreshes to do this.

The emulation was improved by breaking the screen up into tiles of 8x8 pixels. Each tile can be marked as dirty or clean. Initially each tile is marked as dirty. When a write to the frame buffer occurs, the video card frame buffer is updated, and a corresponding tile on the screen is marked as dirty. Whenever a CLUT entry is updated, or the offset register is changed, all the tiles are marked as dirty.

The speedup comes from having a separate routine which is called every 200,000 CPU cycles. It scans every tile, and if any tile is dirty, it is redrawn from the framebuffer, and marked as clean. As well as being much faster than the naïve scheme (tests showed it to be nearly 10 times faster for normal situations, as well as avoiding the pathological case when the CLUT is updated), it is also closer to the real hardware–on the video board, writing to the framebuffer is handled separately from the hardware that repeatedly scans the framebuffer and generates an analogue signal to send to the display.

A further refinement was made to this scheme, to 'smooth out' the cost of updating the screen somewhat. When the screen was refreshed every 200,000 cycles, if all of the screen tiles were dirty, this could take a substantial amount of real time. This could impact badly on timing sensitive parts of the emulation such as the networking. Instead, the scheme was changed to refreshing one row of tiles every 3,000 cycles. This corresponds even more closely to the real hardware–a row of tiles is not dissimilar to a scanline, and also prevents the video refresh code from hogging the (real) CPU for too long.

5.4 A New Model for ELAN over TCP/IP

As a part of the work for this project, a new model was designed for the ELAN networking emulation. There were several considerations taken into account when designing it:

- The networking must have a sockets-like API which allows the programmer to do both blocking and unblocking reads and writes without having to worry unduly about the internal implementation. Blocking calls are useful for utilities like a native filestore, and unblocking calls are needed to hook into an emulator.
- The networking must be modular and have no particular dependancies on the Fred Machine emulation itself. This makes it easy to write lots of quick little utilities that communicate with the filestore or Fred Machine.
- The networking should, if possible, work across multiple real machines.

- The networking should require no manual mapping of IP addresses to host IDs, and allow connected ELAN interfaces to change their Host IDs.
- The networking should support broadcast ELAN packets.
- It should be possible to generate a trace of all network traffic from a centralised location. An ELAN utility similar to tcpdump[13] would help immensely in debugging implementation problems.
- Ideally, it should be possible to implement the networking emulation without requiring pthreads or other forms of concurrency.
- The implementation should avoid inefficiencies such as requiring connections to be opened and closed for each frame transmitted. There is no way of avoiding a network round-trip per frame, because of the way the ELAN acknowledgement works, but we should avoid more than that. As the networking saying goes "bandwidth can be bought, but latency is forever".

Some thought was put into the design, but an implementation was not finished due to time constraints. The intent is to have a library that provides the ability to have multiple virtual ELAN interfaces (i.e. network cards, or stations), each with their own state. The interface state keeps track of the ELAN state machine, including pending transmits and acknowledgements, so the user doesn't have to worry about it.

The protocol works by connecting each ELAN interface to a *hub* process using a persistent socket connection. The hub is responsible for keeping track of which host ID is connected to which socket, and routes incoming frames from one socket to the appropriate destination(s). It also provides a central point where logging can be performed. In effect, the hub process is like a real Ethernet hub, and the socket connections are like the physical cables connecting the hub to the stations.

The API is as follows:

- ELAN interfaces are created by calling `elan_open_st` with a host ID as an argument. If the library succeeds in connecting to the hub and registering the host ID,

it returns a pointer to an `elan_st` structure. The connection to the hub is bidirectional, so the library never needs to use the blocking `listen` call to receive data.

- ELAN interfaces are destroyed by calling `elan_close_st` with the station as an argument. The station is shut down and disconnected from the hub. All pending sends, receives and ACKs are discarded.

- ELAN interfaces can change their Host ID by calling `elan_change_st_id` with the station and the new ID as arguments. The hub allows the ID change if no other connected station has the same ID. If the ID change occurs, the frames queued for transmission are not modified in any way, and keep their old frame headers.

- Ports are opened using `elan_open_port` specifying the station, port, and address the port should be connected to. Sends and receives to that port are restricted to the specified address. Already open ports cannot be reopened[2], and port 0 is always open, by default.

- Ports are closed using `elan_close_port`, specifying the station and port. It discards all pending sends, receives and ACKs. Closing an already closed port returns an error and does nothing.

- Data is sent on an open port by using the call `elan_send` specifying the station, port, data buffer, buffer length and whether the call blocks or not. The length must be less than `ELAN_MAX_DATA_SIZE`. If the call is set to block it doesn't return until an ACK is received, or the send times out, or the port or interface closes.

  The port keeps track of the number of ACKs that have been received by the library, but have not been acknowledged by the software that uses the interface. We call these *incomplete ACKs*. ACKs received for blocking sends are implicitly

[2]This is at odds with the firmware semantics, but can be handled easily by emulation code. We vary from the firmware semantics because this behaviour is safer and less liable to difficult to track down bugs.

acknowledged, and never count as incomplete ACKs. If the call is set to not block, once the ACK is received the incomplete ACK count for the port increases by one.

Only one unblocking send can be in flight at any one time. An attempt to perform a second unblocking send while one is pending acknowledgement causes an error. Attempting a blocking send while an unblocking one awaits an acknowledgement will work–the blocking sends merely waits for the acknowledgement before proceeding with its own send.

Sends to port 0 have an extra 6 address bytes prepended to the data buffer, and the length should include this.

- The emulation can check if there are any *incomplete ACKs* by using the call `elan_sends_completed` with the station and a port number as an argument. This returns immediately with the number of incomplete ACKs. The library internally resets this counter to zero, thus this call provides a means of *completing* these ACKs.

- Data is received on an open port by using `elan_recv` specifying the station, port, a preallocated buffer, and the buffer length. The call blocks: it doesn't return until a frame is received on that port, or the port or interface close. The reception code automatically handles sending the correct acknowledgement to the sender.

- The emulation can check if there is any pending data on any of the ports by calling `elan_has_incoming` with the station as an argument. The call returns immediately with the number of the lowest port with received data, or -1 if none have data. This is non-blocking and is intended for implementing the receive side of the interface between the station and the emulated Fred Machine.

This API is fine for implementing native communications–the user can write simple serial code using the blocking variants of the send and receive calls, and things should work fine.

The emulator can not use blocking calls without tying up the whole emulation, but for receives it can just use `elan_has_incoming` to see if there are pending receives, and if there are, it can be sure that `elan_recv` will return promptly.

Similarly, the emulator can use `elan_send` in non-blocking mode, followed by calls to `elan_sends_completed` to check if it has completed. Due to the nature of the ELAN host/station protocol, this count should never be more than one.

Clearly, working code speaks far more authoritatively than words in a design document, and it is unfortunate that we don't have an implementation to back up the ideas here. It is hoped that an implementation will be completed in the near future.

5.5 Tweaking the Emulation

As with many engineering projects, a working computer depends on a large number of complex components working as designed, and interacting harmoniously. The devil is in the details, and this applies just as much to an emulation as it does to a real machine.

In the emulation, two major sources of 'interesting' problems was the networking emulation and the filestore code. Here were a few bugs that turned up:

- Initially the code would not compile correctly. Evidently, the version of GCC used (version 3.3) provides a stricter and more standards compliant C++ compiler than the GCC previously used. A number of small tweaks, such as defining namespaces, and adjusting the types of variables were required.

- When the nameserver, filestore and emulation first successfully ran, the boot ROM would successfully determine the memory size and read and start the kernel (`FMAC:NSYS`). The kernel would initialise and attempt to load the CLI (`FMAC:ACLI.MOB`). This appeared to succeed, judging by the filestore logs, but the Fred Machine just printed out the numbers '4F 10 74 01 C2 82 2E 81 52 78 10' on separate lines before freezing. Eventually, after tracing through the boot ROM, kernel source, and filestore source it turned out that the filestore was sending back a malformed response to ReadSq requests. Fixing this allowed the boot sequence to continue to the point of running `ASTARTUP.COM`

- After a number of transactions, the filestore would appear to stop sending responses to the Fred Machine. This could be fixed by restarting the filestore. This became most obvious when repeatedly booting the Fred Machine–sufficiently many transactions occurred between the Fred Machine and the filestore to trigger the bug. It prevented further work when the Fred Machine got as far as running `ASTARTUP.COM`. Some investigation uncovered that in both the emulation and the filestore, the networking code was opening a socket for each ELAN frame sent, but never closing it. Since the system in use (MacOS X) enforces a per-process limit of 256 open file descriptors by default, and socket handles count as file descriptors, the processes were eventually running out of file descriptors. Annoyingly, the code for frame transmission was duplicated a few times in the code, but once tracked down, this was easy to fix.

- If the filestore was stopped and restarted in the middle of a transaction, the Fred Machine would retransmit an ELAN frame as a part of the ELAN protocol (since it hadn't received the ACK it expected). This results in the filestore receiving a request for a transaction it has no record for. It correctly logs an error noting this fact, but still attempts to use that transaction number, and crashes. Again this was an annoying problem, but one that was easily fixed once diagnosed.

- An attempt to use the text editor IE froze the Fred Machine because it was trying to open a configuration file called PROFILE.IE. The file did not exist on the filestore, and the filestore code correctly checked for this case, but was sending no reply to the client in response. The client expects some response from the filestore, and ends up waiting indefinitely for one. The filestore was modified to send an appropriately formatted error (of the form `- ; File ANON:PROFILE.IE not found`) to the client.

- Attempting to use the text editor IE exposed the fact that the filestore didn't have any knowledge of what the default directory for a user connected to the filestore was. This caused problems with OpenR requests. Since user logins are not yet implemented, a patch that recognises public users is all that is required in this case.

- Attempts to perform requests that aren't implemented in the filestore resulted in no response from the filestore. This causes the Fred Machine client to freeze waiting for a response. This was fixed by returning a generic `-2 Not implemented` error to the client. Programs that attempt to open files for writing are given the error `-= No authority`.

- The general filestore request has an option to return the current time from the filestore. This was used by a few utilities (such as SUGGBUG) and so was implemented.

- A minimal version of the Quote request (which is used by the game Asteroids) was implemented to allow it to run.

Chapter 6

Results and Conclusion

> There are three arts which are concerned with all things: one which uses, another which makes, and a third which imitates them.
>
> Plato, *The Republic*

The stated aim of this project was to improve the emulation of the Fred Machine so it could run interactive, and graphical software.

This aim was met: The terminal emulation works sufficiently well that buffered Command Line input works correctly. Programs that read and write to the console in an unbuffered mode[1] and use VT220 escape sequences and cursor keys (such as games, and VT220 based menus) do what they should do. Special keystrokes such as Ctrl-Y (exit program) and Ctrl-T (enter the low level debugger) work correctly. Indeed, if the Fred Emulation were run over a serial console (rather than in a software terminal window), its behaviour should be difficult to distinguish from the real thing.

The graphical emulation is accurate and fast enough to allow several games, many graphical demos, test programs, and utilities to run.

The emulated environment has been improved to be more reliable in the face of the increased demands that interactive usage puts on various parts of the system–especially on the filestore.

There is always room for improvement, however.

[1] that is to say one character at a time, rather than one line at a time

It was unfortunate that a new model for ELAN emulation was designed, but insufficient time was available to write a working implementation. A filestore which supported user login, passwords, quotas, and writing to files would also have made the emulation much more complete. The lack of write access to the filestore meant that it wasn't possible to test the Fred Machine's C, PASCAL and IMP compilers, which was a disappointment.

A functioning debugger, with an awareness of the of the Fred Machine OS's memory layout would be very useful, especially if it allowed the user to examine system structures such as the external dictionary, the network buffers, etc..

Another useful tool would be a utility to trace all the network traffic over the ELAN.

Despite these failings, the emulation as it stands emulates enough to give a good 'feel' for the software environment that the Fred Machines provided in Edinburgh throughout the 1980s and early 90s.

Appendix A

Some Fred Machine Hardware

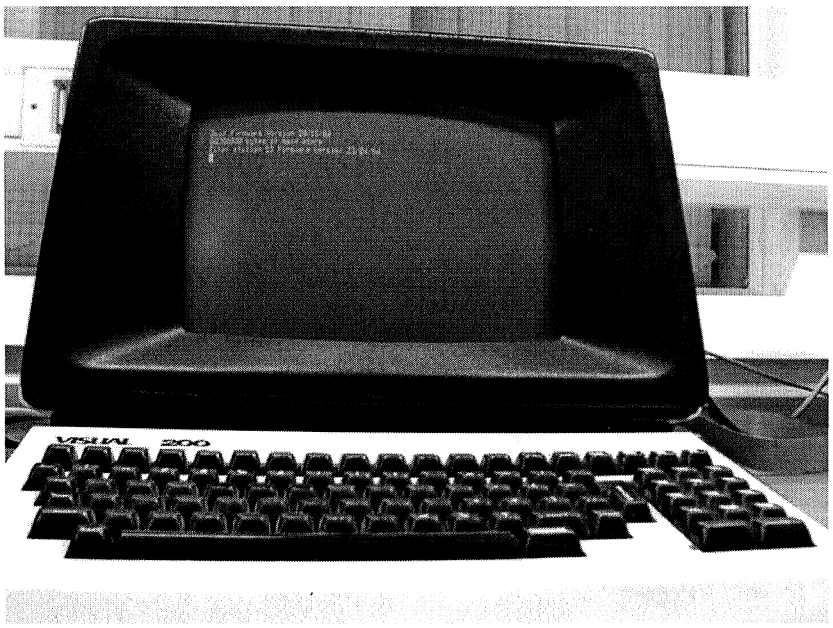

Figure A.1: The Visual 200 terminal

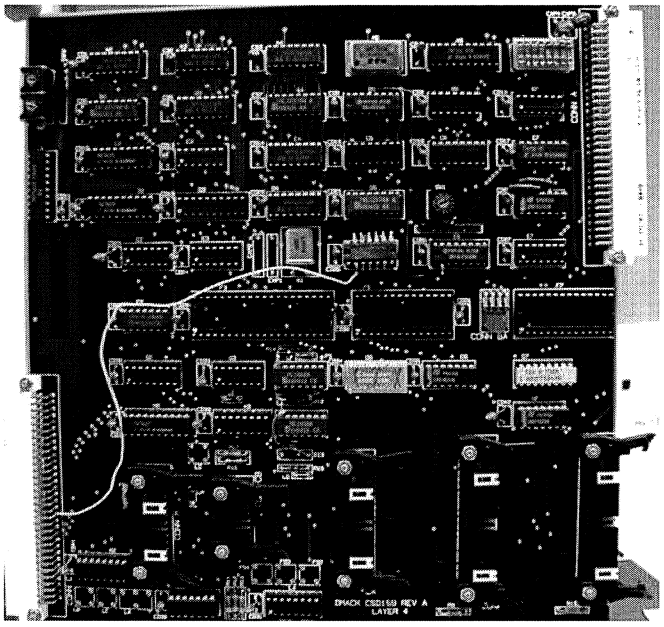

Figure A.2: The DMACK serial controller

Appendix B

Some Fred Machine Software

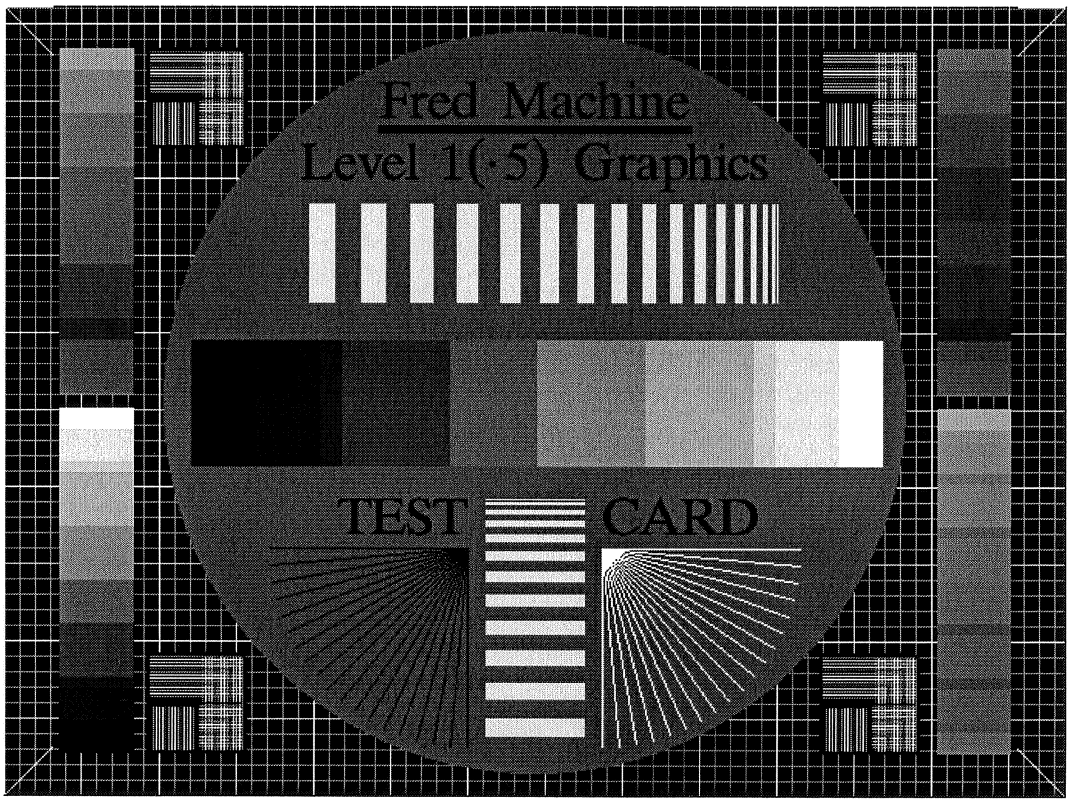

Figure B.1: LEVEL1:TESTCARD

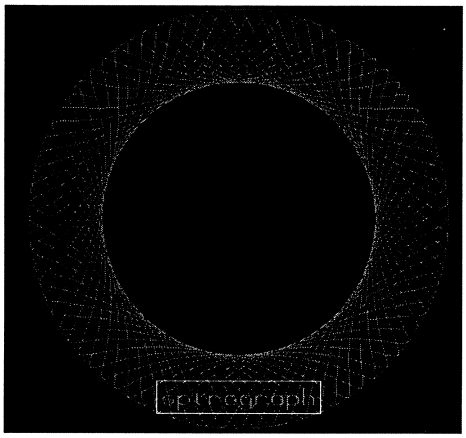

Figure B.2: ADEMO:SPIRO

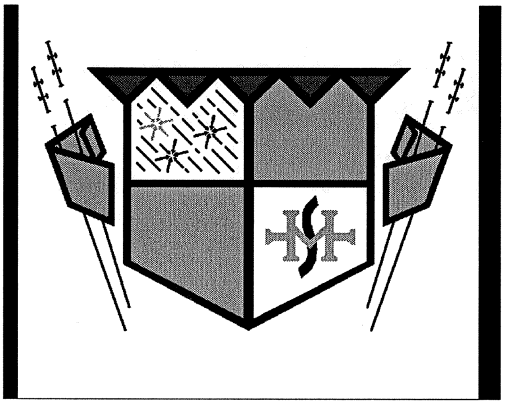

Figure B.3: ADEMO:SPARKLE

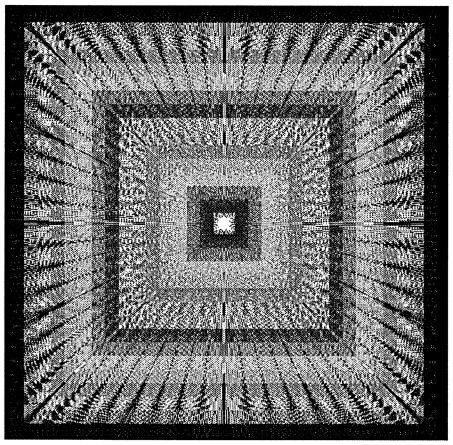

Figure B.4: ADEMO:PERSIAN

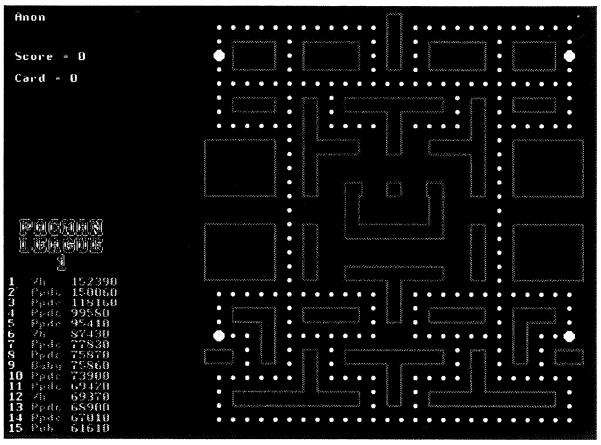

Figure B.5: GAMES:PACMAN

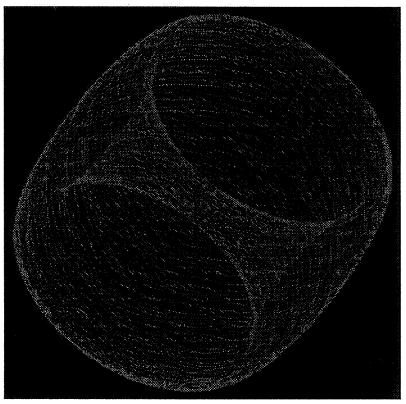

Figure B.6: ADEMO:HARMONY

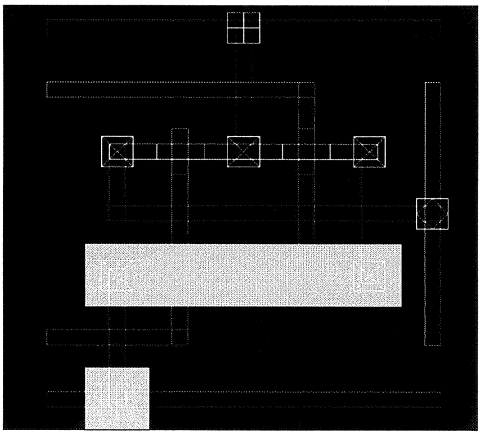

Figure B.7: EDWIN:GILVIEW

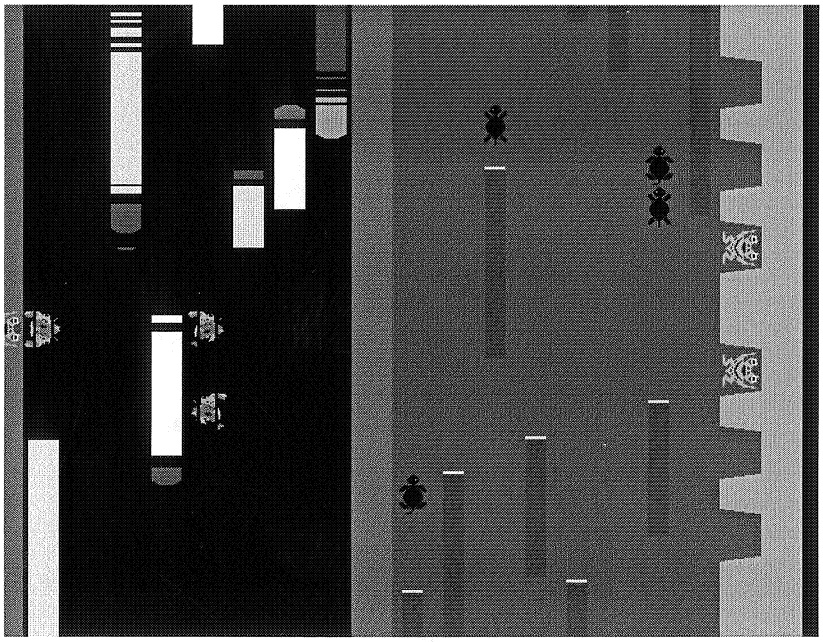

Figure B.8: CP:FROGGER

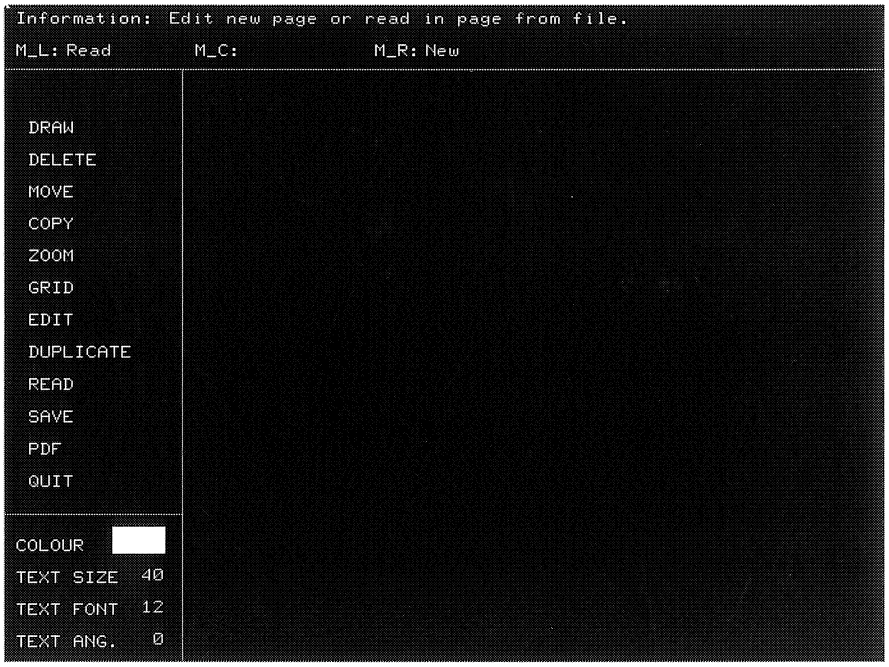

Figure B.9: EDWIN:DRAW

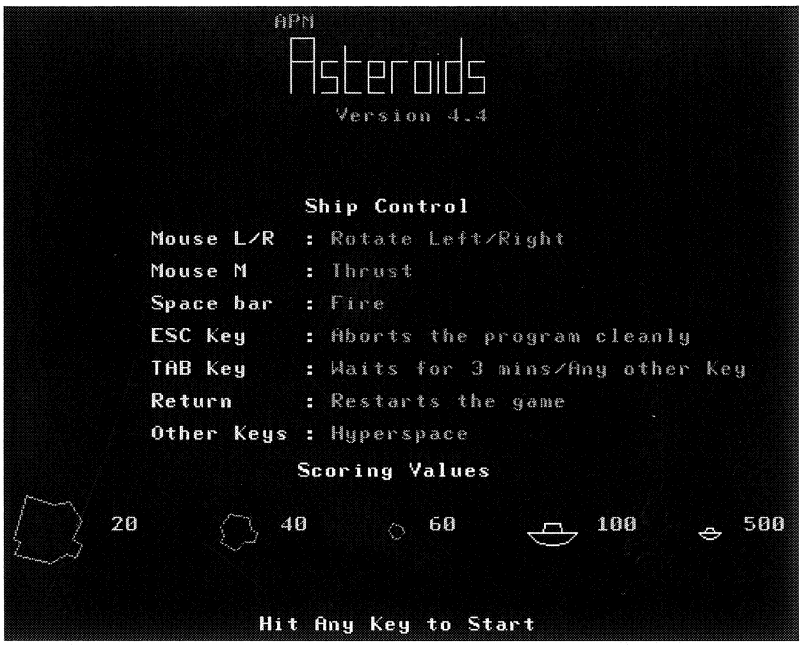

Figure B.10: GAMES:AST

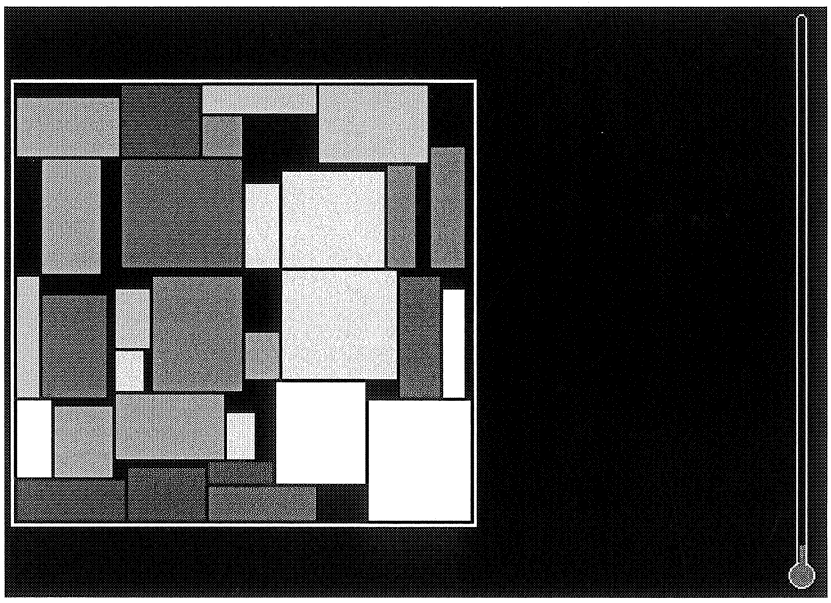

Figure B.11: ADEMO:ANNEAL

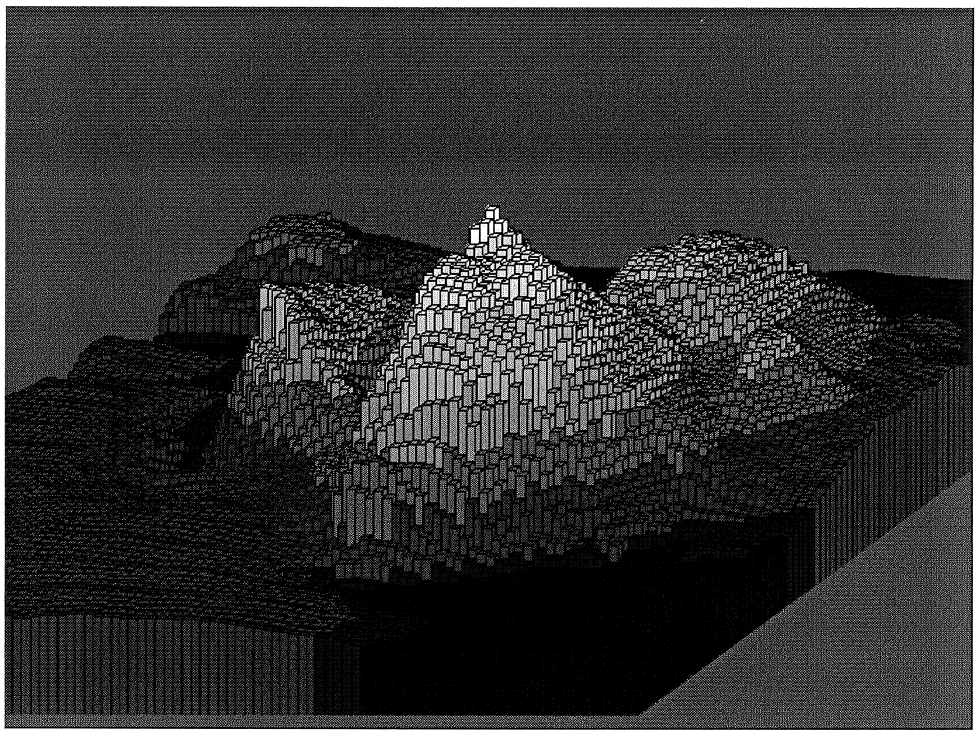

Figure B.12: IFF:3D SOLID:ARTHUR.IFF

Figure B.13: FRED.IFF Fred King, December 3, 1986

Figure B.14: GORDON.IFF (Gordon Brebner, July 1, 1987)

Figure B.15: JHB.IFF (John Butler, October 13, 1986)

Figure B.16: JHBMUG.IFF (John Butler, October 31, 1987)

Bibliography

[1] Christopher Lisle *Fred Machine Emulation* 2001-2

[2] Gordon Brebner *The Evolution of the Fred Machine*

[3] W.P. Enos, I.B. Hansen, R.W. Thonnes *Edinburgh Local Area Network* 1982

[4] Hamish Dewar, Vicki Eachus, Kathy Humphry, Paul McLellan *The Filestore* September 1981

[5] Douglas Rogers *Sons and daughters of APM* December 1988

[6] *APM working documents* May 1983

[7] George D.M. Ross *The (New) Filestores* October 1985

[8] Robert Metcalfe and David Boggs *Ethernet: Distributed Packet Switching for Local Computer Networks* Communications of the ACM, 19:7, July 1976

[9] Charles Thacker, *et al. Alto: A Personal Computer* Xerox PARC, CSL-79-11, 1979.

[10] *The Multiple Arcade Machine Emulator* `http://www.mame.org/`

[11] *VNC–Virtual Network Computing from AT&T Laboratories Cambridge* `http://www.uk.research.att.com/vnc/`

[12] V. Jacobson *et al. RFC 1323 – TCP Extensions for High Performance* `http://www.ietf.org/rfc/rfc1323.txt`

[13] `http://www.tcpdump.org`