I don't care if space aliens ate my mouse

Porting Linux To The Apple Macintosh

Alan Cox
Red Hat, Inc.

alan@redhat.com

A study in both the technical and human issues in porting the Linux OS to a new M68K target platform. And an analysis of the effectiveness of the platform independant features of the Macintosh. Mostly however an essay on the dangers of saying "Well all you need to do is...."

1. Copyright and Licensing

2. Starting Points

There are several barriers to a Linux for Macintosh 68K port. The first of these is that Apple don't want other operating systems on their machines. Whereas you can learn almost all of the workings of a PC from books you will find almost nothing written on the Apple Macintosh. Sometimes the Macintosh specifications and tech notes fill in the blanks at other times its neccessary to apply a great deal of guesswork and experimentation to figure out the hardware.

The second barrier is a human barrier. Most Macintosh machines were not sold to the technical market, and the average Macintosh user isn't terribly interested in a 'real operating system' for their computer. There is nevertheless a sizeable technically oriented Macintosh user community and a lot of Macintosh hardware around (more probably than any other non Intel Linux platform). A further reason has been provided by Apple whose attitude to 68K machines now appears to be 'quaint, buy a new computer'.

The third barrier to a Linux port is less obvious and is hidden by the lack of documentation. Certain folks have speculated that embarrasment is the main reason for Apple Computer releasing so little documentation. The Macintosh platforms in general have positively stone age design features. For example the interrupt controllers on a Macintosh II are a pair of 6522 VIA chips, intended for use with the 8bit 6502 processor. Stupid hardware makes for poor performance unless carefully handled. The complete lack of DMA is even less helpful. Apple seem to think no DMA is a feature on most machines and actually have a technote 'I used to be a teenage DMA junkie' which attempts to justify their rather comical hardware design.

3. Getting Started

So what do you need to get a port started. The first item is hardware. I had most of this (a 5Mbyte MacII cast off from the office as too slow for anything). Initially I felt safe in helping work out the directions for the Linux port as this system lacked an MMU and was therefore unable to run any proposed Linux port.

Rob Pelkey started on some very basic Linux work for the Macintosh but needed a boot loader to load the Linux OS and kick it off. On #linux on the LinuxNet IRC network Jes Sorensen, the keeper of Linux68K, I and several other random people got into a few discussions about the port and what would be required. After a lot of digging we managed to establish some basic information on the Macintosh68K and then fill further areas in by investigating the excellent detective work the OpenBSD/Mac team had done in getting BSD limping along on the Macintosh machines. Further information came to light from the Linux on OSF Mach port that Apple sponsored when we realised that Apple continued to use the same 8bit microcontrollers or emulations of them, and that Apple had not redesigned the systems materially for the new processor.

Everything seemed completely happy. I had a Macintosh box to laugh at (and we used it occasionally to fail to duplicate problems Macintosh users had with CymruNet), we could kick ideas around, and I had no MMU in my Macintosh so I couldn't possible help to write any code.

By this time Rob's effort had stalled badly as he lacked the time to write the boot loader needed to run Linux and was working on passing classes and other sundry items. No worry, someone would eventually take over the project or he would finish his classes. And then Frank Neuman sent me an MMU for the MacII and someone else donated a pair of ethernet cards. Whoops, sudden shortage of excuses.

4. Learning MacOS

Having fitted the MMU to the Macintosh without blowing it up I tried to get

MacOS to run with virtual memory. This is supposed to be simple. You click on the memory tool and select 32bit, virtual memory on. Oh no, my memory control didn't have a 32bit option let alone a virtual memory one. I stared a bit, checked on a more modern mac downstairs to be sure I had the right screen. The other Mac which was running the same MacOS version had the required option, I didn't.

This is when I first learned the horrors of the Mac. While Unix says 'Im sorry you can't do that', MacOS has two error messages. It either goes 'eep?' or the box you wanted to set but couldn't is simply not there on you computer until you've installed the other 12 unidenfied items and filled in 3 apparently unrelated dialog boxes. This was an error of the latter category.

It turns out that Apple shipped the MacII with the ability to upgrade to include an MMU chip. Therefore they sensibly shipped it with a system ROM that wasn't capable of of running with the MMU enabled. Brilliant, just don't design anything mission criticial please. Fortunately Apple had concealed on their web site a small tool which patches the ROM entry points so that it can run in 32bit mode.

Ok so all you do is download the tool, install it and off you go. Not so simple. To get the program onto the machine I needed to get the ethernet to work. I ended up using kermit to transfer 700K of ethernet installer onto the Macintosh. About 4 hours of fighting with the completely alien Macintosh archiver tools I had the machine talking appletalk shares to a Linux box using netatalk and an insight into why Mac people meeting a PC for the first time look like they just discovered alien life forms.

About an hour after that I had figured out how to unpack Macbin files and the Macintosh was in 32bit mode and admitted the MMU was present and functional.

5. Building and Booting Linux

The next stage in the operation was to figure out how to boot a Linux kernel image on the Macintosh. NetBSD and OpenBSD use a boot loader which loads a.out format executables into the memory of the Macintosh, shut the macintosh down, move them to address 0 and jumps to it. I rapidly decided I didn't want to write a boot loader. The OpenBSD loader was almost pure MacOS wizardry at a level far beyond my abilities. Not to worry, it soon became apparent that the OpenBSD loader could be persuaded to load Linux too. A true loader could wait.

The next problem was to build a Linux kernel image that would link and while probably not do anything useful at least serve as something to feed the OpenBSD booter. Linux is built using the GNU toolchain which supports the building of cross compilers. It is thus possible to compile and build 680x0 binaries on an ordinary intel based PC. It took a couple of builds to get gcc and the GNU binutils almost generating the right code. Linux-aout executables have a two byte different header to the OpenBSD ones and the the OpenBSD boot loader checked these bytes. Rather than rebuild the entire toolchain again I wrote a simple tool to fix the headers.

Most of Linux/M68K was quite content to build for a Macintosh target. I filled in everything that complained with dummy routines - for Mac keyboards, mice, display etc until it all compiled. Because of the well designed abstraction layers in the Linux/M68K kernel this is quite easy to do. I now had a completely useless do nothing Macintosh kernel that the OpenBSD loader would load, and which then promptly crashed the Macintosh as I expected.

The Linux/m68K project had faced up to the challenges of supporting multiple types of 680x0 based computer within the same port well before I got involved. As a result of the need to support both the Amiga and Atari systems there are clear layers of abstractions. Adding an additional m68k target consists mostly of filling in platform specific blank fields. A port to a completely new processor would have been far more challenging than this.

For the macintosh case I filled in various mostly blank function handlers. After finally getting the thing to link I ended up with a kernel that hardcoded for a 5Mbyte 68020 based macintosh with FPU and a display at 0xF9000000. It had no interrupt controllers, no disk controllers, no keyboard, no mouse and anything else I could find was also hard coded. But it linked and that was the important item. Having done a bit of reading up on the innards of the console drivers (and much interrogation of Jes) I wrote a fairly simplistic back end for the generic console driver on the Macintosh. As it turns out the very simplistic approach reflected the Macintosh hardware I had, which was a completely unaccelerated bitmapped display supporting 640x480 in 4bit colour.

6. Paint It Black

A Linux 68K kernel starts with a partially shared piece of initialisation code written in 680x0 assembler and using almost all the most gothic and peculiar features of the architecture. This initialisation code also sets up the memory management and caching and touches everything nobody normally knows about. The 68020, 68851, 68881 combination of chips using in the Macintosh II is obsolete and Motorola therefore didn't carry documentation on this device. I knew two things which in theory were enough to debug and figure out what was going on. Firstly I knew the base address of the screen memory, secondly I knew the address that the code would begin executing. The very first routine I put in the startup code painted the screen a revolting blue colour. After about 15 boots and some staring at the source code I had a Macintosh that booted to a blue screen waited a short while and crashed.

In many way this was the single hardest item to get going. When you are dealing with a completely unknown system environment and have no idea what is around your code it is extremely tricky to debug. Real commercial hardware people use logical analysers. I didn't have the option. I learned several things in the process notably that the Macintosh screen memory isn't located where the hardware claims until you set up the MMU. I also made the amazing discovery that the rounded corners on the Macintosh display are drawn in software.

Over a period of the next few weeks the Macintosh went through an assortment of debugging stripes and coloured patterns as I inched a few lines at time through the initialisation assembler code, fixing it bit by bit and gradually mapping in the needed hardware. Eventually the kernel hit the magic start_kernel() function in the C code without crashing on the way.

7. Consoling Yourself

Hitting start_kernel() is in theory the beginning of the easy road. On a PC at least you have text mode consoles instead of stripes, on a Macintosh hitting start_kernel() meant that the prospect of getting the kernel to initialise a text console and begin showing useful debugging information was close. Nothing could have been further from the truth.

After several attempts to get the console up I wrote some routines to print penguins and macs on the screen (this was easier than text). Each significant point the kernel reached added a penguin to the display and a failure point before the console came up printed a given number of burning macintosh logos. While hardly as good as print statements this was good enough to rapidly locate several bugs in the processing of options passed by the boot loader (little things like apparently having 0K of memory tend to upset the Linux memory initialisation). The code would get to the beginning of the console setup and die.

To get past this point I had to fill in support for the 4bit packed pixel displays that were used by the Apple Macintosh ’Toby’ display card. The generic bitmapped console drivers for the 680x0 port supported a wide variety of pixel formats, and naturally excluded the one I needed.

Had I known at the time I could simply have switched the machine to Mono in the display preferences but at the time I didn’t know the physically switched the card into a monochrome mode. Adding 4bit packed pixel wasn’t too difficult. I left the somewhat scarier 2bit packed pixel support for later, in the hope someone else would have to write it not me. The console code is also very modular on the 680x0 and these console layers (abscon, fbcon) are now used by most non Intel ports. It’s reasonable to assume that it will be driving all the ports by the 2.3 kernel series.

The machine still crashed mysteriously and all evidence pointed to a structure getting stamped on. I put guard values either side of it and checked they were not overwritten, I moved the structure in memory and I tried everything I could think of in order to stop it being apparently corrupted. No joy, no change. After a bit of head scratching I added code to check the values were ok at boot, and at initialisation of each subsystem. The value was wrong at the start of the C code. I checked it at the start of the assembler and it was wrong by then.

This was beginning to look worrying, it seemed that the boot loader was corrupting data, yet this made no sense as the loader would corrupt the same location, not pick on a specific helpless little variable wherever it may have been located. Eventually I used the GNU objdump tools to look at the binary I was loaded. It turned out that the GNU linker was at fault and in some places was loading a completely bogus address for a relocation.

A new linker and the magic words 'Calibrating Bogomips' appeared on the screen, followed by a hang, and there was much rejoicing. In many ways the time lost to the linker bug was not that bad. Eyeballing the code in search of the mystery bug I had fixed some twenty or thirty other serious bugs in a vain attempt to find the illusionary real bug.

I wasn't too worried the Bogomip calibration hung. It's very hard to calibrate time before the interrupt routines, especially the timer interrupt routines have been written. I commented it out and after a short while the rest of the code booted to the point of saying 'Panic:unable to mount root filesystem'. A reasonable situation as I had exactly no device support except the screen.

8. Filling In The Blanks

Getting the machine to the point where everything appears to boot this far is actually by no means any kind of completion of the first steps of a porting project. It tends to be the point at which you finally appreciate the real problems and the scale of work remaining.

There are numerous pieces of hardware in an Apple Macintosh and while it is possible to ignore them trying to get to the initial panic about the root filing system I was going to have to fill at least some of them in to go any further.

The most important items to fill in where those that dealt with the most basic system resources - interrupts, memory and the I/O busses. The interrupts and several I/O subsystems are handled by a pair of 6522 VIA chips, 8bit controllers from the stone age. These chips themselves are documented and their locations were known even if some of the connections to their I/O pins were a mystery. A certain amount of mapping work and other detective information showed that the VIA chips provided the all important system timer ticks, handled the keyboard at an extremely low (and at the time undeciphered) level, and provided interfaces for the external interrupts from the bus controllers.

Several other pins appear to do things like turn the Macintosh off. Even now we don't know what everything on the VIA chips does or if all the pins have a real use. It also turned out I got the easy end. The later Macintosh machines replace the second VIA with a device known as RBV (Ram Based Video) which contains a bad emulation of a VIA chip and various other components in one piece of glue logic.

Basic interrupt handling on a Macintosh is relatively clean. A great deal of attention has been paid to keeping interrupts that need a fast response at a higher priority that time consuming processes. That works well under MacOS but Linux itself tends to take rather too binary a view of interrupts especially in the drivers. Certain interrupts are wired in strange ways presumably to save components - the SCSI interrupt for example is wired through a VIA but is effectively upside down compared with the other interrupt sources. Apple saved an inverter by using the fact the VIA can handle either direction of state change as an interrupt signal.

I ended up with two layers of interrupt handling, which were mostly hard coded. Unlike a PC the Macintosh interrupts are very much hard wired. Only the Nubus (plug in) cards change positions, and they all share one interrupt which sets bits in a VIA register to indicate the real interrupt source.

Nubus proved quite entertaining. The documentation is quite weak and all written from the point of view of building a card for a Macintosh. It took about a week before the boot up code would scan and report a list of which nubus slots were occupied and the name of the devices. Once it worked the Nubus turned out to be an extremely well designed system with features much like PCI. Each slot is allocated a set of memory resources and can raise an interrupt. A ROM allows the OS to read each device for identification and driver information. The ROM also contains other "useful" data including icons for the device. At the moment these are not made visible under Linux, but the intention is to support /proc/nubus/[slot]/icon.xpm at some time.

9. Mapping Ethernet Cards

The Daynaport card I had been given was very close to several PC designs. The 8390 ethernet chip and block of RAM on it made that quite clear. There are however 2^24 possible locations for the chip and memory within each Nubus slot space.

Finding where the device was hidden required building a collection of kernels which searched the 24bits of address space looking for two things. Firstly looking for areas of memory which could be read and written, secondly looking for areas like this which had the additional property of giving different results when read back. The 8390 chip has several control registers, and by playing with these it is possible to fairly reliably identify the chip (this same code is used to probe for NE2000 and WD80x3 cards in Linux for PC). On the Macintosh the RAM was easy to find but the 8390 did not show up.

Having played with the RAM behaviour a bit I discovered that the memory was mapped to every alternate 16bits in its address space. That is if you wanted to read it you had to read two bytes, skip two bytes, read two bytes etc. A bit of further experimentation revealed that the Ethernet controller registers occurred every fourth byte, that the RAM occurred every other pair of bytes and was 16bit wide and that the ethernet controller saw the 16bit wide memory as 8bit wide. Only on a Macinotsh...

These sort of techniques work for mapping a large number of devices and address spaces, and helped to discover the location of additional devices in the Apple I/O spaces. We still don't know enough to drive the Apple sound chip and the "Integrated Woz Machine" (floppy disk controller), but we do know where they are located.

10. Rooting For NFS

When you need to start testing a system booting into user space you need a file system. The NFS root file system is extremely attractive for this and has been used for most ports. The NFS (Network File System) makes transaction requests at the level of files rather than disk blocks. This has the saving grace that errors in the new port cause transactions to get rejected. If you are trying to debug a new port and a SCSI controller driver at the same time you will instead spend much of your time reformatting and reinstalling the disk you are attempting to boot from. Using NFS bounds the possibility for errors and also makes it easier to add and edit files as you attempt to make the machine work.

The initial installs were done with a set of tar files for the m68k known as "watchtower". Watchtower is extremely outdated but is small and it was easy to unpack. Since the goal was getting a shell prompt the age of the binaries was not a serious worry. Watchtower also demonstrates another strength of Linux/m68k. All the ports run the same binaries. Instead of having to cross compile and debug all the binaries for the Macintosh I was unpacking and booting a file system set up for installation on a Commodore Amiga.

With a few modifications to the drivers and several small bugfixes to the kernel code the applications started to run. As most of the code you need to add for a new M68K platform is drivers and setup code once things started to work most applications sprang to life. It took a couple of tweaks to get floating point to always behave itself but once done I was able to boot the machine fully multi-user, but without keyboard, mouse or hard disk support.

It took almost a month before anyone else got the kernel to boot on their own machine. A lot of debugging removed some rather bad assumptions that had 'escaped' the code clean up and gradually other MacLinux 68K machines began to pop into being. This is an extremely important step for any project as it allows other people to contribute effectively. Michael Schmitz wrote the SCSI drivers and much of the keyboard and mouse support. He is now adding IDE. Numerous other people have tested and debugged the code on the many varieties of Macintosh, and even made it work on some.

11. Conclusions

While any new port is difficult the structure of the Linux M68K kernel tree is very well designed and delivers on its intention to allow easy portability between M68K targets. Several sections of this code are (rightfully) now being used cross architecture as well as cross platform.

Making a free software port work seems to be about having a small number of people willing to take the project the first 50% of the way. Once you hit this point the project gathers momentum of its own accord. Even when its something is pointless as Linux on a Macintosh II.

Lack of documentation is only a hinderance. It will not stop determined people exercising basic rights to use and operate property they have bought and own. Instead it reflects badly on the vendor who is trying to be a nuisance. If the only documentation on the keyboard interface is entitled 'Space aliens ate my mouse', someone will still find it.

Always be the second operating system port to an undocumented platform. The sterling work done by the OpenBSD/Mac team was a huge help to the Linux project. I'm also happy to say that while half of the world may sit on usenet advocacy groups throwing manure the relationship between the Linux and BSD Macintosh teams has always been one of mutual co-operation. Together we advance our detective work and knowledge of the Macintosh platforms to the good of all Macintosh users dumped and orphaned by Apple.

12. Thanks

Michael Schmitz, Yves, and everyone on the linux-mac68k list who has helped build and test the project.

Rob Pelkey for starting the whole escapade and writing much of the booter.

Frank Neumann for dropping me in at the deep end by donating an MMU.

Jes & Geert for their explanations of the innards of the M68K port and consoles.

The MacBSD team for cracking much of the macintosh before us.

Everyone else who contributed to the Linux/mac68K project however large or small their part. And there are many of them.

Keith Baker at CymruNet whose decision to trash the MacII made all of this possible.

13. No Thanks

Steve Jobs - For refusing to provide any Mac68K documentation

Steve Jobs - For refusing to let anyone else pass on Mac 68K documentation

Steve Jobs - For refusing to provide NeXT documentation to the NeXT project

Steve Jobs - For refusing to let anyone else pass on NeXT cube documentation

Steve Jobs - For killing the Newton

Steve Jobs - For refusing to provide any documentation about the Newton to the Linux ARM project