BootX: The Mac OS X Bootloader

Louis Gerbarg

The Macintosh has used a more or less unchanged boot mechanism for over a decade. Even with initial introduction of OpenFirmware, little changed. The advent of the iMac, and later Mac OS X, has altered the boot sequence significantly. This paper contains a cursory look at OpenFirmware, the booting mechanisms used by various operating systems that run on the Power Macintosh (such as Linux, NetBSD and OpenBSD), as well as the different booting mechanics of several generations of Macintosh hardware. Particular emphasis will be paid to the boot process of Mac OS X (from the firmware up to early kernel initialization) and its bootloader, BootX.

Introduction

Bootloaders load operating systems and provide early boot services such as boot time operating system selection. This paper is an introduction to the bootstrapping environment for PowerPC based Macintoshes focusing on Mac OS X, and its bootloader, BootX. All code examples in this document are covered by the Apple Public Source License (APSL) version 1.2. Text of the APSL can be found at http://www.opensource.apple.com/apsl/.

What is a Bootloader?

A bootloader is a program that is run when a computer is started that is responsible for loading an operating system. The loader may perform a number of actions, but its fundamental responsibility is to place the computer in a state that the operating system can start in.

Why is a bootloader necessary?

Even on systems with advanced firmwares, such as OpenFirmware[1] and Sun's OpenBoot[2] it may be desirable to provide functionality not available in the firmware. It also makes is possible to overcome deficiencies that may exist in particular firmwares.

Who needs to know about BootX?

BootX is useful to developers who intend to use configurations that differ from the default boot settings. This might include those who want to network boot, load operating systems besides Mac OS X, or run BootX on non-Macintosh hardware.

Organization

This paper discusses seven major, interlated topics. The first is an historical view of Macin-

[1] OpenFirmware is referred to extensively throughout this manual. The draft version of the the OpenFirmware specification may be found at ftp://playground.sun.com/pub/p1275/coredoc/

[2] OpenBoot is an OpenFirmware compliant boot prom found in Sun Microsystems computers

tosh bootstrapping. The second is a summary of the boot sequence of various machines. The third is an overview of alternative operating systems for Macintosh hardware, and a brief overview of their bootloaders. The fourth is an overview of OpenFirmware. The fifth section is an over of the the Mac OS X bootloader, BootX. The sixth section is an introduction to the internal structure of BootX. The final section discusses extending and debugging BootX

Macintosh Bootstrapping

The Macintosh has been through a number of significant revisions since its inception. Its bootstrap mechanics and firmware have changed in many significant ways. Within this section we will look at the basic booting mechanics of various revisions of Macintosh hardware.

68k

Though discussing the entirety of the 68k line of Macintoshes is beyond the scope of this paper, understanding how they worked at a basic level is useful. Nubus cards that were used in the boot process would have a ROM with a driver on them. When the machine powered up the Mac OS ROM would read the device ROMs on the cards, and use those to traverse any bootable devices to look for a System Folder.

It is important to note that drivers on those cards were in 68k machine code, and expected certain machine characteristics. This meant that when new machines were released sometimes cards that were bootable on older hardware would require a ROM upgrade to work with a newer machine (which was costly, since most cards did not have Flash ROMs).

Nubus Power Macintoshes

Nubus Power Macintoshes were in many ways 68k Macintosh motherboards modified to work with PowerPC processors. There is a PowerPC ROM that starts up various parts of the system, and brings up the system nanokernel. The nanokernel then starts the 68k emulator, which executes a largely unchanged 68k Mac OS ROM. Overall it tended to work in manner very similar to the 68k machines.

Old World PCI Power Macintoshes

Starting with PCI machines an entirely new ROM was used. Based on work previously done by Sun, these machines use a ROM specification known as OpenFirmware. Old World is a term that Apple coined to refer to their OpenFirmware 1.x and 2.x implementations.

OpenFirmware was not used to its fullest on these machines. The OpenFirmware ROM was used to load another section of the physical ROM, which was very similar to the previous Mac OS ROM. The new Mac OS ROM walked through the PCI devices and ran machine specific PowerPC machine language drivers it found on the cards.

The fact that OpenFirmware was only used to load a fixed ROM meant it was not generally tested, and number of limitations are present when using it that are not present booting through the fixed Mac OS ROM.

New World PCI

Starting with the iMac, Apple released its 3.0 implementation of OpenFirmware, referred to as New World. New World machines removed the fixed Mac OS ROM. Instead their OpenFirmware loaded a file named "Mac OS Rom" from the boot device's System Folder.

This change facilitated a number of improvements. Runtime ROM patches were no longer necessary, since Apple could simply update the "Mac OS Rom" file. Additionally since OpenFirmware was used to load "Mac OS Rom" from a device, OpenFirmware was enhanced to overcome the deficiencies in earlier implementations.

This change was not without its share of problems. A number of PCI devices did not have OpenFirmware FCode Drivers, which meant they required ROM replacements to become bootable on New World machines.

Boot Processes

The PowerPC Nubus Mac OS Boot Process

This is a summary of the classic Mac OS boot process for PowerPC Nubus machines.

1. Machine starts nanokernel
2. Nanokernel starts 68k emulator
3. 68k Mac OS ROM is started
4. 68k Mac OS ROM searches for potential devices
5. 68k Mac OS ROM uses Macintosh specific drivers it finds in device ROMs.
6. 68k Mac OS ROM scans all potential devices for a System Folder (preferably using the default disk stored in PRAM
7. Mac OS ROM starts Mac OS

The Old World Mac OS Boot Process

This is a summary of the classic Mac OS boot process for Old World machines.

1. Machine runs lowlevel initialization
2. OpenFirmware is started
3. OpenFirmware looks for a boot-device (a default value of /AAPL,ROM is stored in the firmware variable boot-device)
4. Mac OS ROM is started (from /AAPL,ROM)
5. Mac OS ROM searches for potential devices
6. Mac OS ROM uses Macintosh PowerPC specific drivers it finds in device ROMs.
7. Mac OS ROM scans all potential devices for a System Folder (preferably using the default disk stored in PRAM
8. Mac OS ROM starts Mac OS

The New World Mac OS World Boot Process

This is a summary of the classic Mac OS boot process for Old World machines.

1. Machine runs lowlevel initialization
2. OpenFirmware is started
3. OpenFirmware looks for a "boot-device" (a default is stored in the firmware)
4. OpenFirmware loads a file of type 'tbxi' ("Mac OS Rom") from the partition
5. Mac OS Rom is started
6. Mac OS Rom starts Mac OS

The Mac OS X Boot Process

This is a summary of the standard Mac OS X boot process.

1. Machine runs lowlevel initialization
2. OpenFirmware is started

3. OpenFirmware looks for a boot-device (a default is stored in the firmware variable boot-device)
4. OpenFirmware loads a file of type 'tbxi' (BootX) from the partition
5. OpenFirmware executes BootX
6. BootX reads root partition out of nvram
7. BootX loads mach_kernel from the device
8. BootX copies Mac OS X device drivers from partition into memory
9. BootX disables all address trnaslations
10. BootX starts Mac OS X mach_kernel
11. mach_kernel begins its boot process
12. mach_kernel may use an integrated linker to link Mac OS X device drivers into itself if it is necessary to complete booting
13. mach_kernel unlinks the integrated linker to save memory

Other Operating Systems on Macintosh hardware

A number of alternative operating systems have been ported to Macintosh hardware, including, but not limited to, Mklinux, Linux, and NetBSD. This section take a brief look at what hardware each of these runs on, and how they are booted.

Mklinux

Mklinux was a port of Linux to the Mach microkernel that was done jointly by the Open Software Foundation, and Apple Computer. Parts of the work involved with this later went into the Apple Mac OS X Mach kernel. Mklinux initially ran on Nubus PowerPC hardware, and later supported a subset of the PCI models. Its default bootloader, mkboot, ran an extension to Mac OS. Because it was run after Mac OS had already been loaded it could boot off of any device that Mac OS could boot off. Unfortunately, it required a copy of Mac OS. Recently, alterations have been made that allow mklinux to work with the Linux BootX bootloader as well.

Linux

There is a port of Linux to the PowerPC platform, and runs on PCI Power Macintoshes. Recently significant progress has been made on a port to Nubus. machines.

Since Linux was initially targeted at PCI hardware it used a small OpenFirmware bootloader, called quik, which ran on Old World machines. Because Linux for the PPC was the first widely available Operating System to use Old World OpenFirmware for booting it also was the first to publicly demonstrate many of the bugs. Recently a much more featured loader for New World OpenFirmware, named yaboot, has become available.

Besides the OpenFirmware based bootloaders there is a Mac OS based bootloader, named BootX (not to be confused with Mac OS X's BootX). BootX runs as either an application or extension to Mac OS or as an application. As such it has basicly the same benefits and downsides as mkboot.

Finally, there is a variant of BootX known as miboot. miboot is uses basicly the same code as BootX, but it sheds the Mac OS specific code. It is installed in fake a System Folder so that the fixed Macintosh ROM will attempt to boot it. It then uses the ROM primitives to load a linux kernel. This has most of the benefits of a Mac OS based bootloader, but it does not require Mac OS, so user is not required to purchase a Mac OS license. This particularly useful for Nubus based machines, which cannot use OpenFirmware based bootloaders. It is also incompatible with New World ROMs, which do not have

a fixed Mac OS ROM.

NetBSD

NetBSD has several options for booting. NetBSDÕs kernel may be compiled as an ELF so that when it is stored on an HFS partition it is directly bootable by New World ROMs. They also have a bootloader named ofwboot that can be used by both New and Old World Macintoshes. NetBSD does not currently run on Nubus machines.

Summary of Alternate Bootloaders

A number of other bootloaders are available for Power Macintosh, though none of them are compatible with nearly as many machines as BootX. Table 1 describes various bootloaders and the hardware they support. The Mac OS column indicates whether the bootloader runs from with a booted Mac OS environment. The Mac OS Rom column indicates if it requires a machine with a fixed Mac OS ROM (Old style Power Macintosh PCI machines).

OpenFirmware

OpenFirmware is the ROM used by current Power Macintoshes. It has a number of features that make one of the most flexible firmwares available.

OpenFirmware has many convenient aspects. It supports the use of machine independent drivers. This means that the same PCI card will work as a bootable device in both Sun UltraSparcs and Power Macintoshes. Additionally, it provides an interface for user configuration, and an interface for programs to access its services.

User Interface

Users can access the OpenFirmware User Interface by turning on the computer while holding cmd-opt-O-F. At this point the user can manipulate various aspects of the firmware. Several useful commands are:

- boot - executes the "boot-command"
- dev - moves to a device node
- dev / - moves to the root node
- go - returns execution to the previous (non-OpenFirmware) environment
- ls - lists all the items in a node
- printenv - displays environment variables
- setenv - sets an environment variable
- see - shows the code for a command
- words - lists all the commands that can be run on a specific node

Additionally there are a number of environment variables that alter the firmwares behavior

- auto-boot? - a boolean value which tell the firmware whether to show its user interface or automaticly execute the "boot-command"
- boot-command - The command to execute when the user enters "boot"
- boot-device - a path to the "boot-device"
- boot-file - a path to a kernel image that BootX will use instead of its defaults
- nvramrc - stores patches that are executed on the firmware
- use-nvramrc? - a boolean value which determines whether or not the firmware will execute a patch (stored in "nvramrc") on itself before executing the "boot-command"

- security-mode - Used to determine the firmwares security settings. See the Security section for more on this
- security-password - A hashed form of the firmware password. printenv has been coded specificly not to display this value.

Besides these function, there are a number of other commands specifically for configuring hardware, as well as all of the standard Forth commands. The interface should be able to run arbitrary Forth commands.

Device Interface

Since OpenFirmware has a Forth interpreter, it is sensible that OpenFirmware drivers are written in Forth. Because OpenFirmware uses a Forth interpreter to run the drivers, the driver can be used on any processor architecture. Additionally to save space in the ROM the drivers are encoded in a byte code representation of Forth, known as FCode.

Device Tree

OpenFirmware devices are mapped into a hierarchal tree, known as the device tree. This is used to navigate to particular devices, and serves as a basis for the IORegistry within Mac OS X. A user can navigate the device tree using the "dev" command within the OpenFirmware User Interface. When a user moves into a particular node they may use commands specific to that device.

Client Interface

OpenFirmware provides an interface for programs it loads to use its resources. This is known as the Client Interface. BootX is dependent on the Client Interface to load Mac OS X. There are a large number of calls available. The Client Interface is not accessible outside of the bootloader under Mac OS X, and is not stable on a large number of Macintoshes. For more information look at the ci.tproj subproject of the BootX source code. The Client Interface becomes unavailable after the "quiesce" command is used, which happens immediately before BootX transfers control to the kernel.

Binary Loading

OpenFirmware is capable of loading a binary from any device it can access. While New World supports loading ELF binaries as well as XCOFF, Old World only supports XCOFF. Additionally some versions of OpenFirmware can load a format known as bootinfo, which is basicly any type of binary OpenFirmware understands with an XML header. Neither firmware can directly boot the Mac OS X kernel, which is in the Mach-o format.

Security

Enabling Firmware Security

With the newest firmware updates for the PowerMac G4, iBook, iMac (slot-loading), and Powerbook (firewire), apple has made their firmware compliant with the security portions of IEEE1275. To enable Firmware security, boot into the OpenFirmware User Interface. Use the "password" Forth word, which will prompt the user to set your password. Once this is complete the user may use setenv to set security-mode to one of three values "none", "command", or "full".

Differences between the security-modes

The firmware's default mode is "none". In this mode it acts just as it has always acted. Once the user sets the security-mode to something else things are very different.

In "command" mode the system will only

boot off the default boot-device (on Macintoshes this generally the one selected in the Startup Disk control panel). It will no longer boot off of CD or Network by holding C or N respectively. In order to execute any OF command besides "boot" or "go" the user will need to enter the firmware password. The user will be unable to zap PRAM. If the user holds down the option key at boot they will prompted for a password before they can use the boot chooser. This mode is intended for lab scenarios, to prevent use from hacking systems by booting of off CDs, and increase overall maintainability.

In "full" mode all the restrictions of command mode remain in effect. In addition the machine will act as the OF variable "auto-boot?" is set to false, forcing the user into the OF user interface. Executing the boot command now requires the firmware password. This mode is intended for situations in which a machine that is forced to shutdown should not be restartable until a proper admin is available.

Emergency password recovery

There is an emergency mechanism to reset the firmware password if the user forgets it. If the machine detects it has a different amount of RAM installed then it did the last time it booted, it will allow the user to zap PRAM, regardless of the current setting in "security-mode". This will cause both "security-mode" and "security-password" to be reset. Once appropriate settings have been restored the machine can be shut down and the memory returned to its original configuration. This is acceptable, since machines that are intended to be secure most have physical security preventing access to their internals.

BootX

Goals

The boot mechanism for Mac OS X had a number of goals. BootX, along with firmware upgrades and nvram patches, fulfills these goals. It provides a robust booting solution for Mac OS X for all of Apple's supported hardware. It supports booting from both of Apple's filesystems, HFS+ and UFS, as well as network booting via tftp. It also understands how to load the Mach-o mach_kernel. Additionally, Apple added a number of features that were not directly necessary for Mac OS X, but which might be useful to other Operating Systems wishing to take advantage of the reliable boot strapping infrastructure it provides. These features include support for the ext2 filesystem, and ELF binary loading to BootX.

It is important to note that alterations to both BootX and other operating system kernels may be necessary in order to make them interoperate. For instance, in addition to the ext2 and ELF support, the Linux kernel will require a new entry point and altered initialization code in order to use BootX.

Requirements

BootX requires a PCI based Power Macintosh, and with some modifications it may work on an OpenFirmware compliant CHRP system. Because many of the OpenFirmware releases are buggy, certain machines will require nvram patches for BootX to function properly. Though they are not technically part of BootX, they may be necessary. Though there has been some preliminary work on necessary patches to overcome bugs in the Power Macintosh 5400/6400/5500/6500 class machines, they are incomplete. Some clones may require patches as well.

Features

BootX contains a number of features available in other bootloaders, and a number that are unique. This section describes its various features.

Binary Formats

Different operating systems use different binary formats. These formats have a number of differences that prevent easy mixing. BootX is capable of loading kernels in both the Mach-o and ELF formats. This allows it to load kernels or second stage loaders for Mac OS X, Linux, OpenBSD, NetBSD, and FreeBSD. In general it can only load staticly linked binaries. Most of these operating systems may require modified kernels with new entry points in order to support BootX.

Volume Formats

BootX understands a diverse series of filesystems. It supports HFS+ (the preferred filesystem of Mac OS), 4.4BSD Big Endian UFS (Mac OS X only, slight incompatible with the other BSDs), ext2 (the preferred filesystem of Linux) as well as loading kernels over any OpenFirmware via tftp.

Security

BootX allows for secure booting by disallowing boot time option selection (such as verbose and single-user modes) if the security-mode variable is set. This variable is used by the firmware to prevent alterations to the machines nonvolatile settings.

Alternate Bootloaders

Table 2 is a listing of various bootloaders and what volume and binary formats they support.

Booting Options

The Mac OS X Bootloader

Verbose

A user can tell the operating system to boot in verbose mode by holding down the command and V keys. Under Mac OS X this replaces the graphical boot logo with a text screen filled with traditional Unix boot messages.

Single-User

A user can tell the operating system to boot in single-user mode by holding down the command and S keys. What this means varies between different operating systems.

Alternate Kernel Location

BootX will boot off an alternate kernel if it finds a device path to a kernel in the OpenFirmware boot-file variable. To find out how to specify such a path please refer to the IEEE1275 OpenFirmware specification.

Other kernel arguments

BootX will pass other arguments to the kernel, such as an alternate root-device. These additional arguments should be part of the "boot-args" variable.

BootX Source Overview

The bootx source is divided primarily into three subprojects. bootx.tproj/ci.subproj/ implements the OpenFirmware Client Interface BootX uses. bootx.tproj/fs.subproj/ has the code necessary for reading supported filesystems. Fi-

nally bootx.tproj/sl.subproj includes the main the source code for bootx, as well as miscellaneous bits that were not large enough to breakout into their own subprojects.

Below are three functions from BootX. The first function, Start(), is the actual function that is called by OpenFirmware (sometimes referred to as an entry point). The second is the Main() function of BootX, which provides an overview of what BootX does. Though the all of the details of the various actions are not present, it serves as a good overview of what is going on. The last function CallKernel(), is last action of BootX.

Note that the in Figure 1 the Start() function does very little. It performs some bit of pointer manipulation to setup a stack that is compatible with gcc's calling conventions, and then enters the main function. This is basically just a small bit of glue to deal with the fact that the environment for the compiled code has not been set up. Normally these sorts of actions would be preformed by an underlying OS, or be done in something like crt0.s (the usual name of the glue file C compilers stick at the front of each binary).

Figure 2 shows the Main() function, which is the heart of BootX. InitEverything() sets up all of the global values, reads some values from the firmware, and prepares the environment that is required for the rest of BootX to work. GetBootPaths() figures out where the kernel should be. DrawSplashScreen() places the OS X Happy Mac on the screen. LoadFile() copies the kernel into memory. DecodeKernel() sets it up to be run. LoadDrivers() loads kexts that might be needed by the kernel at boot, such as video and disk drivers. SetupBootArgs() formats parameters that are passed to the kernel, such as the root device, single-user mode, or verbose boot. Finally CallKernel() starts the kernel, and if all goes well does not return.

When CallKernel(), shown in Figure 3 is entered BootX's work is almost complete. Everything has been setup, and the kernel is ready to go. It first calls Quiesce(), which is a shim around the OpenFirmware "quiesce" command. This command tells OpenFirmware shutdown certain functions that would conflict with an Operating System, such as timers and DMA transactions. After Quiesce() portions of OpenFirmware are disabled. A little bit of house keeping is preformed, and the kernel, located at gKernelEntryPoint is entered. The first argument to the kernel, which ends up on the R4 register, is a pointer to a structure containing a large amount of information, including the entire device-tree, the systems RAM and video configurations, and any flags to be passed to the kernel. The second argument is a signature, the value "MOSX". An operating system can determine it was loaded by BootX by testing the R5 register. If it equals "MOSX" the OS can alter its behavior accordingly.

Extending BootX

BootX is currently considered feature complete for Mac OS X. There are a number of features that could be added to it that might be useful. This section looks at its internal interfaces, and how common extensions would interface with it

Environmental Limitations

Before writing code that will run in the kernel, developers are told they should consider whether it is necessary to place it in the kernel. Likewise, extending the bootloader should only be down if the alterations cannot be achieved in other manners. The boot environment as limited RAM, limited runtime status capabilities, and debugging facilities that are significantly different from other debugging environments.

Adding New Filesystem and Binary Format Support

While BootX must be in a particular binary format, and stored in a particular way, it allows the user to load several different binary format

kernels from several different filesystems. A typical extension to BootX would be to add support for booting off either a new filesystem, or loading a new format kernel.

Adding Binary Formats

Two binary formats are currently supported, elf and Mach-o (though elf is not thoroughly tested). The method for adding a new binary format loader is simple. A single function needs to implement. An example would be loading an a.out kernel is show in Figure 4.

DecodeKernel() will also need to be modified to call DecodeAOUT(). The DecodeELF function in bootx.tproj/sl.subproj/elf.c is a simple example, though it is not tested, and may not be entirely correct. DecodeMacho() is much more complex.

It would seem reasonable to break the Decode functions out into a seperate subproject (like the filesystems) if any more are added.

Adding Filesystems

All of the supported filesystems are contained with bootx.tproj/fs.subproj/. To add a new filesystem three functions have to be written. An example would be adding the capability to load a kernel from a FAT32 (msdos) partition is shown in Figure 5.

Strictly speaking, FAT32GetDirEntry() is not necessary if the machine booting using an mkext (a compressed file containing all of the boot time kernel extensions). In that case FAT32GetDirEntry() simply returns an error. This is how tftp works. Once the functions are written bootx.tproj/fs.subproj/fs.c must be modified to call the new functions.

Internal APIs

This section lists various internal functions:

- CICell Open(char *devSpec);
- void Close(CICell ihandle);
- CICell Read(CICell ihandle, long addr, long length);
- CICell Write(CICell ihandle, long addr, long length);
- CICell Seek(CICell ihandle, long long position);
- void CacheInit(CICell ih, long blockSize);
- extern long CacheRead(CICell ih, char *buffer, long long offset, long length, long cache);

The first five functions are primitives that wrap OF routines. The CICell returned by Open() is a device handle, which is used by all the other functions. The value of addr is a pointer to a buffer, and the value of length is the length to be copied to or from the buffer. Seek() jumps to a specific location in the open device. Finally, Read() and Write() return the actual lengths of the operations, or -1 for failure.

The Cache functions should be used for reading metadata from the filesystem.There are several examples in the fs.subproj.

Debugging BootX

Debugging BootX cannot be done using conventional debuggers, or even the using the kernel debugger, since it runs before anything else is loaded. OpenFirmware provides invaluable resources for debugging. These resources are far more advanced than most other firmwares (such as Intel), and can greatly reduce the amount of time necessary to debug changes to BootX.

It is preferable to use a two machine debugging setup with BootX. On older machines the OpenFirmware console is exported on a serial port, and can be viewed through a terminal emulator on another machine. On newer machines OpenFirmware will export its console over telnet. Apple Technote 2004 describes how to do this.

There are few other useful debugging techniques. Setting "auto-boot?" to false will cause the system to enter the OpenFirmware User Interface by default. Changing kFailToBoot to 0 in include.tproj/sl.h will alter BootX's default behavior on error, so that it will return to OpenFirmware. Finally, calling Enter(), will cause BootX to drop back into the OpenFirmware User Interface. This can be used as a break point. The "dumpl" word will dump some memory, by entering the address, then the length, then "dumpl". By calling printf in BootX immediately before Enter(), the address can be easily determined, and the variable can then be examined and altered from OpenFirmware. Finally typing the "go" command will resume BootX's execution.

Acknowledgements

I would just like to thank everyone at Apple who made OS X, and gave me a topic to write about. I would particularly like to thank Josh de Cesare, for his excellent work on BootX, as well as putting up with my incessant prattling.

Name	Nubus	Old PCI	New PCI	Clone	Mac OS	Mac OS Rom
BootX	No	Yes	Yes	Yes	No	No
BootX (Linux)	Yes	Yes	No	Yes	Yes	Yes
miboot	Yes	Yes	No	Yes	No	Yes
yaboot	No	Yes	Yes	Yes	No	No
ofwboot	No	Yes	Yes	Yes	No	No

Table 1: Supported Hardware

Name	Mach-o	ELF	HFS+	UFS	ext2	tftp
BootX	Yes	Yes	Yes	Yes	Yes	Yes
Boot(Linux)	No	Yes	No	No	Yes	No
miboot	No	Yes	No	No	Yes	No
yaboot	No	Yes	No	No	Yes	No
ofwboot	No	Yes	No	Yes	No	No

Table 2: Supported Volume and Binary Formats

```
static void Start(void *unused1, void *unused2, ClientInterfacePtr ciPtr)
{
        long newSP;

        // Move the Stack to a chunk of the BSS
        newSP = (long)gStackBaseAddr + sizeof(gStackBaseAddr) - 0x100;
        __asm__ volatile("mr r1, %0" : : "r" (newSP));

        Main(ciPtr);
}
```

Figure 1: BootX Entry Point

```
static void Main(ClientInterfacePtr ciPtr)
{
        long ret;

        ret = InitEverything(ciPtr);
        if (ret != 0) Exit();

        // Get or infer the boot paths.
        ret = GetBootPaths();
        if (ret != 0) FailToBoot(1);

        DrawSplashScreen();

        while (ret == 0) {
                ret = LoadFile(gBootFile);
                if (ret != -1) break;

                ret = GetBootPaths();
                if (ret != 0) FailToBoot(2);
        }

        ret = DecodeKernel();
        if (ret != 0) FailToBoot(4);

        ret = LoadDrivers(gRootDir);
        if (ret != 0) FailToBoot(5);

#if 0
        ret = LoadDisplayDrivers();
        if (ret != 0) FailToBoot(6);
#endif

        ret = SetUpBootArgs();
        if (ret != 0) FailToBoot(7);

        ret = CallKernel();

        FailToBoot(8);
}
```

Figure 2: BootX Main() Function

```
static long CallKernel(void)
{
	long msr, cnt;

	Quiesce();

	printf("\nCall Kernel!\n");

	msr = 0x00001000;
	__asm__ volatile("mtmsr %0" : : "r" (msr));
	__asm__ volatile("isync");

	// Move the Execption Vectors
	bcopy(gVectorSaveAddr, 0x0, kVectorSize);
	for (cnt = 0; cnt ¡ kVectorSize; cnt += 0x20) {
		__asm__ volatile("dcbf 0, %0" : : "r" (cnt));
		__asm__ volatile("icbi 0, %0" : : "r" (cnt));
	}

	// Move the Image1 save area for OF 1.x / 2.x
	if (gOFVersion ¡ kOFVersion3x) {
		bcopy((char *)kImageAddr1Phys, (char *)kImageAddr1, kImageSize1);
		for (cnt = kImageAddr1; cnt ¡ kImageSize1; cnt += 0x20) {
			__asm__ volatile("dcbf 0, %0" : : "r" (cnt));
			__asm__ volatile("icbi 0, %0" : : "r" (cnt));
		}
	}

	// Make sure everything get sync'd up.
	__asm__ volatile("isync");
	__asm__ volatile("sync");
	__asm__ volatile("eieio");

	(*(void (*)())gKernelEntryPoint)(gBootArgsAddr, kMacOSXSignature);

	return -1;
}
```

Figure 3: BootX Finale

```
long DecodeAOUT(void)
{
        //The file has been read from disk and stored at kLoadAddr contains
        //Determine if the file is a.out, if not return -1
        //Otherwise decode kernel and load in memory
        //Set KernelEntryPoint to the start of the decode kernel
        //return 0
}
```

Figure 4: BootX Binary Interface

```
long FAT32InitPartition(CICell ih)
{
        //Test if the device referenced by ih is a FAT32 partition
        //If not return -1
        //Setup any variables (or cache) necessary
        //return 0
}


long FAT32LoadFile(CICell ih, char *filePath)
{
        //Read the file from filePath on device ih
        //Copy into memory at kLoadAddr
}

long FAT32GetDirEntry(CICell ih, char *dirPath,
                        long *dirIndex, char **name,
                        long *flags, long *time)
{
        //Return information about the nth file in directory dirPath, where n is dirIndex
        //name returns the filename
        // flags returns the file type ( kUnknownFileType,
        //kFlatFileType, kDirectoryFileType, kLinkFileType)
        //time returns the modification date of the file
        //Return 0
}
```

Figure 5: BootX Filesystem Interface