Porting BSD UNIX to a New Platform

Lawrence Kesteloot

June 28, 1995

1 Introduction

The source code of the Berkeley Software Distribution (BSD) UNIX system is freely available. This document is a guide for those who wish to port this system to a new architecture. Porting will most likely involve writing device drivers for the new machine, and may even require rewriting the assembly language portion of the operating system if BSD UNIX has not yet been ported to the target CPU. This guide will discuss how to get started in the project, what files need to be modified, what drivers need to be written, and what steps should be followed when releasing the port for public use.

It is assumed that the reader (along with everyone else involved in the port) has a thorough understanding of C, a good knowledge of how to use UNIX, and has taken at least an undergraduate-level course in operating systems.

1.1 A Brief History of UNIX and BSD

UNIX was developed at Bell Labs (AT&T) in the early 1970's[1]. Several universities bought it along with a license for the source code, and the University of California at Berkeley rewrote and enhanced much of the system from 1979 to 1994[3]. They eventually released the BSD version of UNIX, but it was legally encumbered by the AT&T copyright that still covered parts of the source. Users of BSD had to purchase an expensive source license from AT&T.

In 1991, Berkeley's Computer Science Research Group (CSRG) removed the parts of BSD 4.3 that had been written by AT&T and released the remainder as the Berkeley Networking Release 2 (Net/2). Bill and Lynne Jolitz rewrote the missing pieces, completed Net/2's mostly-complete port to the i386, and released 386BSD 0.0. This was followed immediately by a slightly less buggy 386BSD 0.1, and an indefinitely delayed bug-free 0.2 release caused several groups to start their own BSD release. The FreeBSD group took 386BSD, added enhancements, and have been releasing BSD for the IBM PC line of machines regularly. The NetBSD group decided to release a research version of 386BSD, and they now have ports to more than half a dozen machines. The NetBSD system is best suited to handle different architectures and it is recommended that new ports be based on NetBSD's source tree. The author and two other students ported NetBSD to the Macintosh, and that port is now released as part of the standard NetBSD distribution.

1.2 Project Outline

Porting BSD will probably involve the following steps. You will

- look through the NetBSD ports and find the one that's closest to the machine you are porting to (section 2.1);
- write a stand-alone kernel to test device drivers and loaders (section 2.3);
- either modify or rewrite the assembly language portion (section 3.1);
- finish the basic device drivers and machine-dependent code (section 4); and
- incorporate your changes back into NetBSD (section 5.6).

Eventually you may want to release your port to the public, which will involve making a binary distribution and a set of installation tools, as well as having a mailing list, a FAQ, and a lot of patience (section 5).

1.3 Terms

A few terms are defined here to avoid confusion later. The *kernel* is a single piece of code that is always present when UNIX is running and is responsible for the interface to the hardware and the management of resources such as the CPU, memory, and disks. A *user program* (commonly referred to as a *user-land* program in BSD circles) is any program other than the kernel; that is, any program that can be invoked by the user. This covers programs like `ls` and `cc` as well as daemons such as `inetd` and `timed`.

Part of the kernel is *machine-independent code* and part of it is *machine-dependent code*. The former is shared among the ports and the latter is different for each port. Some code is a hybrid of the two, such as the Motorola 68000-family code that is shared between only some ports.

The *memory management unit* (MMU) is a piece of hardware that translates virtual addresses to physical memory addresses and handles protection between process spaces. Sometimes the MMU is a separate chip that sits between the CPU and memory, and sometimes it is built into the CPU directly. There are *MMU tables* in memory that guide the MMU in its mapping.

There are two kinds of memory addresses. A *physical address* is used to access memory directly; that is, it is the address sent to the RAMs when accessing data. A *virtual address* goes through the MMU first and is translated to a (usually different) physical address. If the virtual address does not correspond to a physical address or if the operation on that section of memory is not allowed (*e.g.*, writing to memory that was mapped read-only), then the MMU will *fault*, causing the kernel to load a page from disk, allocate one from scratch, or terminate the program that caused the fault.

Any executable binary program (both kernel and user) has three parts: the text, the data, and the bss[1]. The *text* is the machine code and is usually mapped read-only so that the MMU will catch errant pointer writes. The *data* is the initialized data of a program and is stored with the text to make up the executable file. The *bss* [2] (Block Started by Symbol) is not saved on disk and accounts for the uninitialized global variables of a program. In UNIX, when a program is loaded, space for the bss is allocated and initialized to zero. Although the program has no guarantee that the bss (and hence its uninitialized variables) is set to zero, this behavior of UNIX is not likely to ever be changed because most user programs would stop working. [3] Be warned, however, that some other operating systems (*e.g.*, MS-DOS) do not

[1]These are short for text segment, data segment, and bss segment, although they are rarely written with the "segment".

[2]The abbreviation "bss" is, for some reason, always written in lower-case.

[3]There are rumors that IBM's AIX clears the bss to the hex value 0xDEADBEEF.

clear the bss and assumptions about its contents at program startup make programs non-portable.

1.4 Resources

The kernel of NetBSD is roughly 150,000 lines and can be a bit daunting to tackle. Most people who first try to hack the kernel get what we call kompernelphobia, which is the feeling of being overwhelmed by the size and complexity of something. Drawing a map of the kernel or writing down the calling sequence can help in seeing an overall picture of what the kernel is doing. This is especially useful in the assembly part of the kernel, where, for example, it may not be obvious how memory is being initialized. When reading kernel code, it is helpful to have the editor be able to jump to the definition of a identifier. The program `ctags` can be used to build a file that the editor can use to follow identifiers.

There are currently around a dozen people actively involved in porting NetBSD to various machines, and it is important to use them as a resource when you get stuck. This is both because they may have an answer to your question and because you may be approaching the problem the wrong way. The four core members of NetBSD can be reached through e-mail at `core@NetBSD.ORG`. The members of the team that ported NetBSD to the Macintosh can be reached at `alice@acm.vt.edu`.

2 Bootstrapping the Project

The term *bootstrapping* comes from the idea of pulling yourself up from your bootstraps and refers to the process of starting something with no foundation. Bootstrapping anything is an interesting task because, by definition, one is starting with almost nothing at all, or at least not enough to directly get what is eventually desired; the foundation must be built incrementally. The steps involved in bootstrapping a BSD port are outlined in this section.

2.1 Picking a Starting Point

It is not practical to write all of the machine-dependent code from scratch. You should use as a base the NetBSD port for the machine most similar to the target machine. Ideally, there should already be a port to the CPU of the target machine; this will save you the trouble of rewriting all of the machine-language portion of the machine-dependent code. The easiest CPU to port to is the Motorola 68000-series, starting with the 68020. There are several ports already for this chip (*e.g.*, Macintosh, Sun 3, Amiga, and HP 300), and a good bit of code is already shared among them. Other supported CPUs are the Intel i386, the MIPS R3000 (whose port name is "PMAX"), and the Sun Sparc chip. Send e-mail to `core@NetBSD.ORG` to get an up-to-date list of working and in-progress ports.

Choose a name of your port that describes as specifically as possible the machine that the port will run on. For example, our Macintosh port was originally called "Mac", but was later renamed to "Mac68k" because the PowerPC chip was not supported. Make a copy of the `/usr/src/sys/arch/`*name* sub-tree (where *name* is the name of the base port you choose), rename the subdirectories and files appropriately, and you are ready to start the port.

2.2 The Cross-Development Environment

When the port is up and running, compiling the kernel will be done from within BSD. Until then, however, a cross-compiling environment will have to be setup on another UNIX platform. It is important that the machine and the operating system be as similar to the target machine and BSD as possible to avoid problems such as different CPU endianess (ordering of bytes in a word), include file discrepancies (*e.g.*, the macros in `ctype.h` are often implemented differently), and object file format differences.

The BSD project uses an enhanced make called `pmake`. Among other things, it allows makefiles to include other makefiles, a heavily-used feature throughout the system. It is the only program in the system that can be compiled with regular make, so it should be built first. The following tools should then be compiled: `gcc`, `as`, `ld`, `ar`, `ranlib`, and `size`.

One possible problem when compiling the development tools is that the native include files and libraries will be used with BSD sources. This may not only cause compilation errors, in which case a few BSD include files may have to be substituted for the native ones, but some library routines may have a different behavior or implementation. For example, the `chmod()` call on our cross-development platform interpreted its parameters differently than the BSD programs expected, so all of the files that the BSD linker generated had only execute permission instead of read and write permissions.

Since the UNIX used for cross-development is likely to already have tools with the above names, it will be useful to write a script that allows you to switch between using the native and the BSD executables. The script should also switch between the native and BSD include files and libraries. Alternatively, the BSD utilities could be renamed and the BSD makefiles and paths updated accordingly.

2.3 The Stand-alone Kernel

Operating systems can boot in one of two ways. On some architectures (*e.g.*, the HP 300), booting the machine causes the ROM to perform some diagnostics, read the first block of the disk into memory, and execute it. This first block usually reads the next eight blocks, which comprise the stand-alone kernel, and executes them. The stand-alone kernel can do very little except read a file from the main filesystem and possibly boot from a network connection. The stand-alone kernel will look through the directory structure of a particular disk partition (the boot partition) for a file of a certain predefined name, such as `unix`, `vmunix`, or `netbsd`, load it into memory, clear the bss, and execute it. On other architectures (*e.g.*, the Macintosh), the kernel is loaded directly by an application (the booter) running in the native operating system (*e.g.*, MacOS). In this case, the stand-alone kernel is not used and its function is moved into the booter (see Section 2.4).

In either case, the stand-alone kernel is a great place to test device drivers and other experimental code without interference from interrupts, memory mapping, or the rest of the kernel. During the Macintosh port, we wrote and tested the display, serial port, keyboard, clock, and harddisk drivers before linking with any kernel code. We also tested some MMU mappings and interrupt behaviors. Another advantage is that the stand-alone kernel, being much smaller than the UNIX kernel, compiles faster, which results in faster development cycles.

2.4 The Booter

Some ports (*e.g.*, the Macintosh) boot directly from the native operating system. In the case of the Macintosh, this takes a bit longer than booting directly at machine startup, but allows the booter to pass in information about the machine that it gets from MacOS (*e.g.*, the location of video memory). A booter must do several things:

1. Given a disk and partition number, find the root directory and the kernel in it;

2. load the kernel (text and data) into a contiguously allocated space; and

3. copy the code of a routine after the kernel and execute the routine.

The routine should disable interrupts, copy the kernel text and data to location zero, clear enough space for the bss, and jump to the start location in the kernel. The routine should be careful not to clear itself when clearing the bss. This can be guaranteed by writing the routine sufficiently high in memory so that its location is at least the size of the kernel's code, data, and bss.

The booter is also responsible for passing information into the kernel. This can be done through registers if the amount of information is small (*e.g.*, debug mode, root disk) or through some more elaborate scheme (*e.g.*, array of text variables) if more data needs to be passed, such as date and time, video address and mode, or machine configuration information.

3 Machine-Dependent Code

The kernel is divided into two major parts: the machine-independent code and the machine-dependent code. The former is shared among the ports, is entirely in C, and contains the algorithms and data structures that do not depend on the architecture on which the kernel is running. The latter contains the routines and modules that are called by the machine-independent code. The assembly portion of the kernel is kept in one file, `locore.s`. The file `pmap.c` interfaces the kernel's machine-independent virtual memory code with the machine's memory mapping facilities. Finally, the miscellaneous machine-dependent code is appropriately placed in `machdep.c`.

3.1 locore.s

The file `locore.s` contains all of the assembly-language code for the port. This includes the interrupt and exception handlers, the bootstrapping code, the task-switching code, and any routines that are small and used often enough that they should be written in assembly (*e.g.*, routines to copy blocks of memory). As little as possible should be written in assembly, since it is not portable or easily maintained. Several ports have broken parts of `locore.s` out into `Locore.c`.

The object file for locore is linked first in the kernel executable and usually includes the vector table for the interrupts. All of these vectors point to stubs in locore that clean up the stack and call the C routine to handle the interrupt or trap properly. Depending on the architecture, the assembly language code might also have to determine the cause of the interrupt. For example, a page fault produces different actions if it was caused by kernel code or user code, or if the accessed memory refers to a swapped-out page or an invalid location.

The code at the label `start:` is jumped to directly from either the booter or the stand-along kernel. It usually turns off interrupts, disables caches, sets up the MMU tables, enables the MMU, enables the interrupts, and jumps to `main()`, the C routine responsible for initializing the machine-independent code.

Most architectures can call a C routine to setup the MMU routines instead of doing it in assembly. On some architectures, however, this code must be in assembly because of the memory layout. For example, on HP 300 machines, RAM is mapped at the end of the physical address space. That is, the last addressable byte is at address $2^{32} - 1$. This means that when the kernel is loaded at the bottom of physical memory, the lowest address (and therefore the address of all variables and routines in the kernel) is dependent on the amount of memory installed in the machine. Since this is not known at link time, the kernel must use the MMU to map itself to virtual address 0. Setting up the MMU tables happens before the MMU is enabled, so all references to variables and routines must explicitly take into account the physical location at which the kernel was loaded. This is not easily done in C, so the code is written in assembly.

The assembly language part of the task-switching code saves and restores machine state (including that of the MMU and floating-point unit). The rest is done in C in the machine-independent code.

3.2 pmap.c

The file `pmap.c` handles the machine-dependent interface to the MMU. The machine-independent virtual memory (VM) code, which is in the directory `/usr/src/vm`, keeps a record of the current mapping for each process. Whenever the mapping changes, the VM code calls routines in pmap to update the MMU state to reflect changes in the kernel data structures. The most common operation in pmap is to add a page to an existing mapping, but sometimes a map must be copied for a `fork()` or erased and regenerated for an `exec()`. Much of pmap for a new port can be based directly on pmaps of other ports to machines with similar CPUs and MMUs.

3.3 machdep.c

The file `machdep.c` contains all of the miscellaneous machine-dependent routines. This mostly involves initializing the system console, allocating memory for kernel structures, handling the machine-dependent part of process signals, dumping kernel core, and handling kernel panics, traps, interrupts, timers, parity errors, different CPUs (*e.g.*, subtle differences between the Motorola 68020 and 68030), odd memory mappings, and debugging output.

3.4 Interrupt structure

Interrupt structure refers to the precedence of interrupts. Hardware events that cause interrupts are assigned CPU interrupt levels. The CPU can disable interrupts of a certain level and below, thus allowing an important interrupt to preempt an interrupt of lower priority, but not vice-versa. Most machine architectures do not allow the software to reconfigure the interrupt structure. If the precedence of interrupts can be reconfigured, though, make sure that the clock that is used for task-switching has its interrupts serviced at a high priority so that lower priority events (such as disk access) are not capable of disabling the preemtive scheduling. The serial port should also have a high priority so that high baud rates can be reliably used.

The precedence of interrupts is important because disabling interrupts is a common way to provide mutual exclusion in the kernel. For example, a routine that modifies tty structures (those that handle input and output to a terminal) will call `spltty()` (for Set Processor Level TTY) just before a critical section and later will reset the level to what it was before `spltty()` was called. On most machines, this call will disable any interrupt that would call a function that handles the tty structures and, in addition, will disable any interrupts with a priority lower than that of "tty". An interrupt structure not well suited for UNIX might cause relatively long interrupt code (*e.g.*, disk access or sound driver processing) to block more frequent interrupts (*e.g.*, timer, keyboard, or serial port). On the Macintosh port, heavy use of the hard disk would cause the system time to be off by several hours a day, and scrolling of the display, being in a tty device, would preempt the keyboard routine and cause keystrokes to be lost. On A/UX, Apple's System V UNIX for the Macintosh, a beep causes serial port data to be lost.

3.5 Noncontiguous System Memory

By default, the kernel assumes that memory is physically mapped as one contiguous block. Some machine architectures, though, have a non-contiguous physical memory mapping, and there is code in the BSD kernel to handle this. For example, the IBM PC has RAM starting at physical memory 0, but the ROM and I/O space are mapped at 640k. Consequently, additional memory must start after the ROM, at physical location 1024k.

If non-contiguous memory is to be supported, the preprocessor constant `MACHINE_NONCONTIG` must be defined, and three functions must be added to the `pmap.c` module. The function `pmap_next_page()` should return the physical starting address of the "next" page of memory. The first time this function is called, it should return the lowest physical memory location. The next time it is called, it should return the starting address of the next page (usually 4096 bytes higher), and so on, making sure to jump over holes in physical memory. `pmap_page_index()` should return a page number given its physical address and `pmap_free_pages()` should return how many free pages are left. See the i386 and Macintosh ports for example implementation of these three functions.

4 Device Drivers

Writing the device drivers for a machine can be either the easiest or the hardest part of a port, depending on how much documentation is available for the machine's hardware.

In BSD there is no strict definition of the role of a device driver, but in general a driver should do the very least necessary to interface to the hardware. The designer of a device driver (and, in fact, any kernel module) should ask himself what features of the driver belong in the kernel and which belong in a user program. Although this decision is easy for some programs (*e.g.*, X Windows or a serial port driver), some kernel modules have arguments for being in both places. For example, SL/IP (the Serial Line Internet Protocol) is implemented in the kernel in BSD, but one might ask whether all users should carry the weight of a module used only by a few people and that could be implemented almost as efficiently as a daemon. File systems, especially seldom used ones, could similarly be implemented as user-level daemons called by the kernel.

On the Macintosh, the video display is implemented as a bitmap and there is no hardware support for fonts. Our video driver (ITE) got quite large with support for fast font drawing, different font sizes, different video cards, different display sizes, VT220 emulation, mouse pointer, cut and paste, virtual terminals, scroll-back, etc. After a long argument among members of the team, it was finally decided that this driver, being similar in spirit to a windowing program, did not belong in the kernel. All the code (except the very basic support for a small font and VT220) was removed and put in a program called `dt` (for desktop), which users could run after having logged in. This gave users all the features they had when the code was in the kernel, and it allowed the code to be swapped out if necessary, to be configured at run-time, to be more easily developed and debugged, and to use more memory for its scroll-back pool. It was a clear win to move it out, although this was not so evident when it was first implemented in the kernel. Any addition to the kernel should be scrutinized and justified in order to avoid kernel bloat, because the speed gained by linking it in is rarely needed.

The following subsections are presented roughly in the order that their drivers should be written.

4.1 Serial Port

The serial port driver is useful because it will help in the initial debugging of the kernel. Serial port drivers are easy to write on most machines because they involve only initializing the interface, sending a byte, and receiving a byte. It is a good idea to initialize the serial port driver early in the boot process (at the beginning of locore after the `start:` label) so that diagnostic output can be printed out at boot time. Since the serial port driver is a tty driver, and thus a bit tricky to write from scratch, it is easiest to copy another port's serial port driver and modify the small bits of code that change with new hardware.

4.2 ITE

Most computers are attached to a video monitor that can be used as the system console. This monitor is either bitmapped or character mapped, but in either case, the kernel is responsible for displaying characters and emulating a simple terminal, such as VT220. The file `ite.c` (internal terminal emulator) is usually used for this purpose. Machines with a bitmapped screen can use code from the Amiga port (if the target machine has hardware support for graphics operations) or the Macintosh port (if it does not). If the target port has a character-mapped screen, then the code from the i386 port is the best choice for a starting base. Code for VT220 emulation can be copied from almost any port and is fairly easy to implement as a finite state machine.

4.3 Disk (SCSI)

SCSI-based machines have two levels of disk code in the kernel. The file `scsi.c` handles the low-level SCSI messages and commands, while the file `sd.c` (SCSI disk) handles the partitions, blocks, and order of disk accesses. The NetBSD project recently moved most of the high-level SCSI disk code into a machine-independent directory, so it is likely that a new port will only have to write the adapter-specific code.

Most SCSI systems can either work by interrupts or by polling. With the former, the SCSI system will make a request to the disk, record that the request was made, then leave the CPU free for other processes. An interrupt from the SCSI controller will cause the SCSI system to figure out what request was just terminated and transfer the data, if necessary. With the latter, the SCSI system must busy-loop until the controller handles the request; this can waste quite a bit of CPU time. A first-cut SCSI system should be written with polling to minimize possible bugs, but an interrupt-driven system should eventually be implemented if the controller supports it.

4.4 Keyboard

Writing a keyboard driver has fairly low priority because the serial port driver can function as a console during the early part of the port. If the interface to the keyboard hardware is simple, then the driver can go directly into `ite.c` with the screen driver. Usually the data returned from the hardware are key codes that must be translated into ASCII, taking into account whether the Shift, Control, or Caps Lock keys are being held down. In addition, it is occasionally useful to support undocumented key combinations that perform special tasks, such as rebooting the machine or displaying the system process table.

If the keyboard has an Alt or Command key, then it can be handled like a Meta key. For example, if the user hits Alt-E, then the keyboard driver could simulate the keys `<ESC><E>`. This is useful for programs like Emacs that use the Escape key to simulate a Meta key.

5 Making a Product

Frederick Brooks says that making a system program is three times harder than making a program and that making a system product is three times harder than making a system program[2]. Since a released BSD system is a system product, it should require nine times as much work as a regular program! Not only do the developers have to write the installation tools and documentation, but they must deal with users who have incompatible hardware or who did not read the documentation and installed the software incorrectly. This section outlines a few tools that are necessary when releasing a BSD port.

5.1 Newfs

A user will start with a computer on which he presumably already has an operating system running. He will need to make a new disk partition for BSD (at least 40 to 50 megs), one for swap (at least twice the size of physical memory), and format the BSD partition. Making partitions is best left to the tools that came with the native operating system, but formatting is likely to require a custom program. The source code to the `newfs` program is in the BSD distribution tree, and making a program to format a partition consists of porting newfs to the native operating system. There are few suggestions that can be given here—the problems encountered will depend on the similarity between the native operating system and BSD.

5.2 Installer

Each port can decide how it wants to distribute its binaries, but it is easiest to use the scripts that are supplied by the NetBSD group. These scripts generate a set of tar files that can be selectively downloaded and installed to set up a working tree. For example, only `bin.tar`, `sbin.tar`, and the kernel need to be installed to see if the system will boot, since `init` is in `/sbin` and the shell is in `/bin`. This is especially useful for users who are not sure if the port will work on their configuration.

Two alternatives are commonly used to install tar files. If a floppy disk driver is written and the kernel can fit on a single disk, then the user can boot from a diskette, mount a memory filesystem (RAM-disk), and untar the files from a set of floppy disks.

If no floppy disk driver is available, then a program running in the native operating system will have to write directly to the BSD partition. This is a *lot* of work, since basically the whole UFS system will have to be ported to the native operating system. (UFS is Berkeley's fast file system, now being replaced with FFS, Berkeley's enhanced fast file system.) Once this is done, though, a fancy install utility can be written that allows copying files in and out of the BSD filesystem, untarring, uncompressing, and other minor operations.

A third choice, if the native operating system allows it and if there is a filesystem in BSD to read the native operating system's partition, is to have the installer load and run a kernel right from the native partition. This is clumsy and, to my knowledge, has not been tried, but would allow the installer to use the UNIX tools directly without having a floppy disk driver or having to port UFS. (This approach might get into a chicken-and-egg problem of the kernel not knowing how to mount the native partition until it has a read a file from it.)

5.3 FTP site

The distribution files and documentation will have to be made available on an FTP site that allows anonymous access and has space for the compressed tar files (roughly 20 megabytes). Make sure to have a file called `README` that clearly explains where to get further documentation and whom to contact for questions.

Currently, the code in BSD that encrypts passwords is not exportable outside the United States. The script that generates the distribution files keeps all of the encryption code in a separate tar file. The rest can be downloaded by anyone, and users missing the encryption tar file will have to store their passwords in plain text. Make sure to clearly say both at the FTP site and in the README file that it is illegal for those outside the United States to download the encryption tar file. For example, this is the notice on `sun-lamp.cs.berkeley.edu`, NetBSD's home site:

> EXPORT CONTROL NOTE:
>
> Non-U.S. ftp users are required to follow U.S. export control restrictions, which means that if you see some DES or otherwise controlled software here, you SHOULD NOT copy it. Look at README.export-control (in / and most other directories) to learn more. (If the treaty between your country and the United States did not require you to respect U.S. export control restrictions, then you would not have Internet connectivity to this host. Check with your U.S. embassy if you want to verify this.)

5.4 Mailing list

Once users start downloading the distribution, they will want to be kept informed of changes and updates. The NetBSD group keeps a set of mailing lists and you can ask them to create a new list for your port. Anyone will be able to mail to `port-`*name*`@NetBSD.ORG` (where *name* is the name of the port) to ask questions or help others.

5.5 FAQ

Most new users will have the same set of questions, so it is useful to keep a list of Frequently Asked Questions (FAQ) on the FTP site. The mailing list software on `NetBSD.ORG` can send out a short message when users subscribe; this message should point to the FAQ to reduce the number of repeated questions on the list. The FAQ should have the date when it was last updated, a table on contents, and a pointer to the distribution FTP site for those who obtained the FAQ from another source and want to insure that they have the most recent version.

The Linux community has recently started using SGML (the Standard Generalized Markup Language) for their documentation. Documentation written in SGML can automatically be converted to plain ASCII, troff, HTML, LaTeX, and Texinfo format. Look on `sunsite.unc.edu` in the directory `/pub/Linux/docs` for the conversion software.

5.6 Intergration and Enhancements

The ported code should eventually be installed in the official NetBSD tree; contact `core@NetBSD.ORG` for instructions on how to do this. In addition, it will be useful to keep up with the changes that other ports make to their machine-dependent code so that your port may benefit from their ideas and bug fixes. Subscribe to the `source-changes` mailing list by sending a "help" message to `majordomo@NetBSD.ORG`.

References

[1] Bach, Maurice J. *The Design of the UNIX Operating System,* Englewood Cliffs, New Jersey: Prentice Hall, 1986.

[2] Brooks, F. *The Mythical Man-Month: Essays on Software Engineering,* Reading, Massachusetts: Addison-Wesley Publishing Company, 1975.

[3] Leffler, Samuel J., M. Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System,* Reading, Massachusetts: Addison-Wesley Publishing Company, 1989.