1 Introduction

One of the features that made UNIX successful was the ease with which it could be implemented on new architectures. This advantage has its down side, which is very evident when you compare UNIX with a single-platform operating system such as MS-DOS: since UNIX runs on so many different architectures, it is not possible to write a program, distribute the binaries, and expect them to run on any machine. Instead, programs need to be distributed in source form, and installation involves compiling the programs for the target hardware. In many cases, getting the software to run may be significantly more than just typing **`make`**.

What is porting?

It's difficult to make a clear distinction between *porting* and *building*. In this book, we'll use three terms:

- *building* a package is the planned process of creating an installable software package. This is essentially the content of Chapter 5, *Building the package*.
- *installation* is the planned process of putting an installable software package where users can use it. This is what we talk about in Chapter 9, *Installation*.
- Some people use the term *porting* to describe a software installation requiring undocumented changes to adapt it to a new environment, not including the process of configuration if this is intended to be part of the build process. Although this is a useful definition, it contains an element of uncertainty: when you start, you don't know whether this is going to be a build or a port. It's easier to call the whole process porting, whether you just have to perform a simple build or complicated modifications to the source. That's the way we'll use the term in this book.

The effort required to port a package can vary considerably. If you are running a SparcStation and get some software developed specifically for SparcStations, and the software does not offer much in the way of configuration options, you probably really *can* get it to run by reading the sources onto disk, and typing **`make`** and **`make install`**. This is the exception, however, not the rule. Even with a SparcStation, you might find that the package is written for a different release of the operating system, and that this fact requires significant modifications. A more typical port might include getting the software, configuring the package, building the

package, formatting and printing the documentation, testing the results and installing files in the destination directories.

How long does it take?

It is very difficult to gauge the length of time a port will take to complete. If a port takes a long time, it's not usually because of the speed of the machine you use: few packages take more than a few hours to compile on a fast workstation. Even the complete X11R6 windowing system takes only about 4 hours on a 66 MHz Intel 486 PC.

The real time-consumers are the bugs you might encounter on the way: if you're unlucky, you can run into big trouble, and you may find yourself getting to know the package you're porting much more intimately than you wish, or even having to find and fix bugs.

Probably the easiest kind of program to port is free software, that is to say, software that is freely redistributable. As a result of the ease of redistribution, it tends to be ported more frequently and to more platforms, so that configuration bugs get ironed out more evenly than in commercial software. Porting a product like *bison*[*] from the Free Software Foundation is usually just a matter of minutes:

```
$ configure
checking how to run the C preprocessor
... messages from configure
$ make
... messages from make
$ make install
```

On an Intel 486/66, *configure* runs for 15 seconds, *make* runs for about 85 seconds, and *make install* runs for about 5 seconds—all in all, less than two minutes. If everything were that simple, nobody would need this book.

On the other hand, this simple view omits a point or two. *bison* comes with typeset documentation. Like most products of the Free Software Foundation, it is written in *texinfo* format, which relies on TEX for formatting. It doesn't get formatted automatically. In fact, if you look for the target in the *Makefile*, you'll find that there isn't one: the *Makefile* ignores printed documentation. I consider this a bug in the Makefile. Never mind, it's easy enough to do it manually:

```
$ tex bison.texinfo
tex: not found
```

This is a fairly typical occurrence in porting: in order to port a package, you first need to port three other, more complicated packages. In fact, most ports of *bison* are made in order to compile some other product, such as the GNU C compiler. In order to get our documentation printed, we first need to port TEX, which is appropriately depicted in its own printed documentation as a shaggy lion. This is definitely a non-trivial port: TEX consists of dozens of different parts, the source tree varies greatly depending on where you get it from, the whole thing is written in Web, Donald Knuth's own private dialect of Pascal, and once you get it to run you

* *bison* is a parser generator, compatible with *yacc*.

discover that the output (deliberately) does not match any printer available, and that you need a so-called printer driver to output it to your favourite laser printer—yet another port.

Under these circumstances, it wouldn't be surprising if you give up and rely on the online documentation supplied with *bison*. *bison* has two different online reference documents: a *man* page and something called *info*, a cross-linked documentation reader from the Free Software Foundation. The man page is two pages long, the info runs to over 200K in five files. There are no prizes for guessing where the real information is. But how do you run *info*? Simple: you port the GNU *texinfo* package. This time it's not quite as bad as porting TEX, but it's still more difficult than porting *bison*.

This scenario is fairly typical: you set out to port something simple, and everything seems to be fine, and then you find that a minor part of the port can really take up lots of time. Typically, this is the point where most people give up and make do with what they have achieved. This book is intended to help you go the whole distance.

Why we need to port

There are three main reasons why a port might be more than a simple recompilation:

- Different operating system. Depending on what features the operating system offers, the program may need to be modified. For example, when porting a program from UNIX to DOS, I will definitely have to do something about file naming conventions. If I port a System V.4 program to BSD I may find I need to replace STREAMS calls with sockets calls.
- Different hardware. This is obvious enough with something like a display driver. If the driver you have is designed for a Sun workstation and you're porting it to a PC, you will be involved in some serious rewriting. Even in more mundane circumstances, things like the kind of CPU involved might influence the program design.
- Local choices. These includes installation pathnames and cooperation with other installed software. For example, if I use the *emacs* editor, I may choose to use the *etags* program to cross-reference my source files; if I use *vi*, I would probably prefer to use *ctags*. Depending on the C compiler, I may need to use different compilation options. In many cases, this seems to be similar to the choice of operating system, but there is a significant difference: in general, changing your kernel means changing your operating system. You can change the C compiler or even the system library without changing the basic system.

Unix flavours

UNIX spent the first ten years of its existence as the object of computer science research. Developed in Bell Labs (part of AT&T), it was significantly extended in the University of California at Berkeley (UCB), which started releasing significant updates, the so-called Berkeley Software Distribution (BSD) in 1977. By the time AT&T decided to commercialize UNIX with System III in the early 80's, the fourth BSD was already available, and both System III and System V drew heavily from it. Nevertheless, the differences were significant, and

despite the advent of System V.4, which basically just added all features available in any UNIX dialect into one package, the differences remain. A good overview of the relationship between the Unixes can be found on page 5 of *The Design and the Implementation of the 4.3BSD UNIX Operating System* by Sam Leffler, Kirk McKusick, Mike Karels and John Quarterman. In this book I will concentrate on the differences that can be of importance when porting from one flavour to another.

Research UNIX

Research UNIX is the original UNIX that has been developed inside Bell Labs since 1969. The last version that became widely available was the *Seventh Edition*, in 1978. This version can be considered the granddaddy of them all*, and is also frequently called *Version 7*. In this book, I'll make frequent references to this version. Work on Research UNIX continued until 1993, by which time it had reached the Tenth Edition. It's unlikely that you'll have much to do with it directly, but occasionally ideas from Research UNIX trickle into other flavours.

Berkeley UNIX (BSD)

The first Berkeley Software Distribution was derived from the 6th edition in 1977 and ran on PDP-11s only. 2BSD was the last PDP-11 version: 2.11BSD is still available for PDP-11s, if you have a need (and a UNIX source licence). 3BSD was derived from 2BSD and the 7th edition—via a short-lived version called 32V—in 1979. Since then, BSD has evolved relatively free of outside borrowings. With the closure of the Computer Science Research Group in Berkeley in autumn 1993 and the release of 4.4BSD in early 1994, the original BSD line has died out, but the public release of the complete sources will ensure the continued availability of Berkeley UNIX for a long time to come.

Current BSD systems include BSD/OS (formerly called BSD/386), 386BSD, NetBSD and FreeBSD. These were all originally ports of the BSD Net-2 tape, which was released in 1991, to the Intel 386 architecture. These ports are interesting because they are almost pure BSD and contain no AT&T licensed code. BSD/OS is a commercial system that costs money and supplies support; the other three are available free of charge. It is not clear how long all three free versions will continue to exist side-by-side. 386BSD may already be dead, and the difference between NetBSD and FreeBSD is difficult to recognize.

At the time of writing, current versions of BSD/OS and FreeBSD are based on 4.4BSD, and NetBSD is planning to follow suit.

XENIX

XENIX is a version of UNIX developed by Microsoft for Intel architectures in the early 80s. It was based mainly on the System III versions available at the time, though some ideas from other versions were included and a significant amount of work was put into making it an easier system to live with. Not much effort was put into making it compatible with other versions of UNIX, however, and so you can run into a few surprises with XENIX. SCO still markets it,

* In fact, a number of UNIX flavours, including System V and BSD, can trace their origins back to the Sixth Edition of 1976, but they all benefitted from modifications made in the Seventh Edition.

but development appears to have stopped about 1989.

System V

System V was derived from the 6th and 7th editions via System III, with a certain amount borrowed from 4.0BSD. It has become the standard commercial UNIX, and is currently the only flavour allowed to bear the UNIX trademark. It has evolved significantly since its introduction in 1982, with borrowings from Research UNIX and BSD at several points along the way. Currently available versions are V.3 (SCO Open Desktop) and V.4 (almost everybody else).

System V.3 lacked a number of features available in other Unixes, with the result that almost all V.3 ports have borrowed significantly from other versions, mainly 4.2BSD. The result is that you can't really be sure what you have with System V.3—you need to consult the documentation for more information. In particular, vanilla System V.3 supports only the original UNIX file system, with file names length limited to 14 characters and with no symbolic links. It also does not have a standard data communications interface, though both BSD sockets and System V STREAMS have been ported to it.

System V.3.2 is, as its name suggests, a version of System V.3. This version includes compatibility with XENIX system calls. As we saw above, XENIX went its own way for some time, resulting in incompatibilities with System V. These XENIX features should be supported by the kernel from System V.3.2 onwards. SCO UNIX is version V.3.2, and includes STREAMS support.

System V.4 is the current version of System V. Previous versions of System V were often criticized for lacking features. This cannot be said of System V.4: it incorporates System V.3.2 (which already incorporates XENIX), 4.3BSD, and SunOS. The result is an enormous system which has three different ways to do many things. It also still has significant bugs.

Developing software under System V.4 is an interesting experience. Since the semantics of System V.3 and BSD differ in some areas, System V.4 supplies two separate sets of libraries, one with a System V personality and one with a BSD personality. There are no prizes for guessing which is more reliable: unless you really need to, you should use the System V libraries. When we discuss kernel and library differences in Part 2 of the book, the statement "This feature is supported by System V.4" will mean that the System V library interface supports it. The statement "This feature is supported by BSD" also implies that it should be supported by the BSD library interface of System V.4.

OSF/1

OSF/1 is a comparatively recent development in the UNIX market. It was developed by the Open Systems Foundation, an industry consortium formed as a result of dissatisfaction with AT&T's policy on UNIX. The kernel is based on CMU's Mach operating system, a so-called *microkernel*[*]. The original Mach operating system was styled on Berkeley UNIX. OSF/1 attempts to offer the same functionality as System V, though inevitably some incompatibilities

* A microkernel operating system is an operating system that leaves significant operating system functionality to external components, usually processes. For example, device drivers and file systems are frequently implemented as separate processes. It does not imply that the complete system is any smaller or less functional than the monolithic UNIX kernel.

exist.

POSIX.1

POSIX is a series of emerging IEEE standards applying to operating systems, utilities, and programming languages. The relevant standard for operating systems is IEEE 1003.1-1990, commonly called POSIX.1. It has also been adopted by the International Standards Organization (ISO) as standard ISO/IEC 9945.1:1990.

POSIX.1 defines the interface between application programs and the operating system, and makes no demands on the operating system except that it should supply the POSIX.1 interface. POSIX.1 looks very much like a subset of UNIX. In fact, most users wouldn't notice the difference. This makes it easy for UNIX operating systems to supply a POSIX.1 interface. Other operating systems might need much more modification to become POSIX.1 compliant. From a UNIX viewpoint, POSIX.1 does not supply as rich a set of functions as any of the commercially available UNIX flavours, so programming to POSIX specifications can feel somewhat restrictive. This matter is discussed in the *POSIX Programmer's Guide* by Donald Lewine.

Despite these slight disadvantages, POSIX has a great influence on operating system development: all modern flavours of UNIX claim to be POSIX-compliant, although the degree of success varies somewhat, and other systems are also attempting to supply a POSIX.1 interface. The trend is clear: future UNIX-like operating systems will be POSIX-compliant, and if you stick to POSIX features, your porting problems will be over. And I have a supply of bridges for sale, first come, first served.

Other flavours

It doesn't take much effort to add a new feature to a kernel, and people do it all the time. The result is a proliferation of systems that mix various features of the leading products and additional features of their own. On top of that, the release of kernel sources to the net has caused a proliferation of "free" operating systems. Systems that you might well run into include:

- *AIX*, IBM's name for its UNIX versions. Current versions are based on System V.3, but IBM has stated an intent to migrate to OSF/1 (IBM is a leading member of the OSF). Compared to System V, it has a large number of extensions, some of which can cause significant pain to the unwary.
- *HP-UX*, Hewlett Packard's UNIX system. It is based on System V.3, but contains a large number of so-called *BSD extensions*. Within HP, it is considered to be about 80% BSD-compliant.
- *Linux*, a UNIX clone for the Intel 386 architecture written by Linus Torvalds, a student in Helsinki. It has absolutely no direct connection with traditional UNIX flavours, which gives it the unique advantage amongst free UNIXes of not being a potential subject for litigation. Apart from that, it has a vaguely System V-like feeling about it. If you are porting to Linux, you should definitely subscribe to the very active network news groups (`comp.os.linux.*`).

- *SunOS* is the generic name of Sun Microsystems' operating systems. The original *SunOS* was derived from 4.2BSD and 4.3BSD, and until release 4.1 it was predominantly BSD-based with a significant System V influence. Starting with version 5.0, it is a somewhat modified version of System V.4. These later versions are frequently referred to as *Solaris*, though this term properly applies to the complete system environment, including windowing system (OpenWindows), development tools and such, and does not apply only to the System V based versions. Solaris 1.x includes the BSD-based SunOS 4.1 as its kernel; Solaris 2.x includes the System V.4-based SunOS 5.x as its kernel.
- *Ultrix* is DEC's port of 4.1BSD and 4.2BSD to the VAX and MIPS-based workstations. It is now obsolete and has been replaced by OSF/1.

I would have liked to go into more detail about these versions of UNIX, but doing so would have increased the size of the book significantly, and even then it wouldn't be possible to guarantee the accuracy: most systems add functionality in the course of their evolution, and information that is valid for one release may not apply to an earlier or a later release. As a result, I've made a compromise: nearly all UNIX features were introduced either in BSD or System V, so I will distinguish primarily between these two. Where significant differences exist in other operating system—SunOS 4 is a good example—I will discuss them separately.

Where does this leave you with, say, NonStop UX version B30? NonStop UX version B is a version of UNIX System V.4 that runs on Tandem's Integrity series of fault-tolerant MIPS-based UNIX systems. It includes some additional functionality to manipulate the hardware, and some of the header files differ from the standard System V.4. In addition, it includes a minimal carry-over of BSDisms from the System V.3 version. Obviously, you can start by treating it as an implementation of System V.4, but occasionally you will find things that don't quite seem to fit in. Since it's a MIPS-based system, you might try to consider it to be like SGI's IRIX operating system version 5, which is System V.4 for SGI's MIPS-based hardware. Indeed, most IRIX 5.x binaries will also run unchanged on NonStop UX version B, but you will notice significant differences when you try to port packages that already run on IRIX 5.x. These differences are typical of a port to just about every real-life system. There are very few pure System V.4 or pure BSD systems out there—everybody has added *something* to their port. Ultimately, you will need to examine each individual problem as it occurs. Here is a strategy you can use to untangle most problems on UNIX systems:

- Interpret the error messages to figure out what feature or function call is causing the problem. Typically, the error message will come from the compiler and will point to a specific line in a specific file.
- Look up the feature or call in this book. Use the description to figure out what the original programmer intended it to do.
- Figure out how to achieve the same effect on your own system. Sometimes, I recommend a change which you can make and try the program again. If you're not sure how your system works, you can probably find a manual page for the feature or call, and this book will help you interpret it.

- Reconfigure or change the code as necessary, then try building again.

Where you fit in

The effort involved in porting software depends a lot on the package and the way it is maintained. It doesn't make much difference whether the software is subject to a commercial license or is freely available on the net: the people who write and maintain it can never hope to port it to more than a fraction of the platforms available. The result is that there will always be problems that they won't know about. There is also a very good chance that the well-known and well-used package you are about to port may never have been ported quite that way before. This can have some important consequences:

- You may run into bugs that nobody has ever seen before in a well-known and well-used package.
- The package that you ported in ten minutes last year and have been using ever since has been updated, and now you can't get the @&*(&@$(to compile or run.

This also means that if you do run into problems porting a package, your feedback is important, whether or not you can supply a fix. If you do supply a fix, it should fit into the package structure so that it can be included in a subsequent release.

To reiterate: it makes very little difference here whether we are talking about free or licensed software. The players involved are different, but the problems are not. In many ways, free software is easier, since there are fewer restrictions in talking about it (if you run into problems porting System V.4, you can't just send the code out on the net and ask for suggestions), and there's a chance that more people will have ported it to more platforms already. Apart from that, everything stays the same.

But can I do it?

Of course, maybe your concern is whether you can do it at all. If you've never ported a program before, you might think that this is altogether too difficult, that you'll spend days and weeks of effort and confusion and in the end give it up because you don't understand what is going on, and every time you solve a problem, two new ones spring up in its place.

I'd like to say "Don't worry, with *this* book nothing can go wrong", but unfortunately things aren't always like that. On the other hand, it's easy too overestimate the things that can go wrong, or how difficult a port might be. Let's look at the bad news first: in most cases, you can assume that the worst thing that can happen when you try to port a package is that it won't work, but in some unfortunate cases you may cause your system to panic, especially if you are porting kernel software such as device drivers. In addition, if you are porting system utilities, and they don't work, you could find that you can no longer perform such essential system functions as starting or shutting down the system. These problems don't occur very often, though, and they should not cause any lasting damage if you religiously back up your system (you *do* perform regular backups, don't you?).

Apart from such possible dangers, there is very little that can go wrong. If you are building a package that has already had been ported to your platform, you should not run into any problems that this book can't help you solve, even if you have negligible background in programming and none in porting.

How to use this book

The way you approach porting depends on how difficult it is. If it's a straightforward business, something that has been done dozens of times before, like our example of porting *bison* above, it's just a matter of following the individual steps. This is our approach in the first part of this book, where we look at the following topics:

- Getting the software. You might get the sources on tape, on CD-ROM, or by copying them from the Internet. Getting them from this format into a format you can use to compile them may not be as simple as you think. We'll look at this subject in Chapter 2, *Unpacking the goodies* and Chapter 3, *Care and feeding of source trees*.
- Configure the package for building. Although UNIX is a relatively well defined operating system, some features are less well defined. For example, there are a number of different ways to perform interprocess communication. Many packages contain alternative code for a number of operating systems, but you still need to choose the correct alternative. People often underestimate this step: it seems simple enough, but in many cases it can be more work than all the rest put together.

  Configuration is a complicated subject, and various methods have evolved. In Chapter 4, *Package configuration*, we'll look at manual configuration, shell scripts, and *imake*, the X11 configuration solution.
- Build the package. This is what most people understand by porting. We'll look at problems running *make* in Chapter 5, *Building the package*, and problems running the C compiler in Chapter 6, *Running the compiler*.
- Format and print the documentation, which we'll investigate in Chapter 7, *Documentation*.
- Test the results to make sure that they work. We'll look at this in Chapter 8, *Testing the package*.
- We'll discuss how to do installation correctly, accurately and completely in Chapter 9, *Installation*.
- Tidy up after the build. In Chapter 10, *Where to go from here*, we'll look at what this entails.

Fortunately, almost no package gives you trouble all the way, but it's interesting to follow a port through from getting the software to the finished installation, so as far as is possible I'll draw my examples in these chapters from a few free software packages for electronic mail and Usenet news. Specifically, we'll consider Taylor *uucp*, the electronic mail reader *elm*, and C news. In addition, we'll look at the GNU C compiler *gcc*, since it is one of the most

frequently ported packages. We'll port them to an Intel 486DX/2-66 machine running BSD/386 Version 1.1.*

Part 2

As long as things go smoothly, you can get through the kind of port described in the first part of this book with little or no programming knowledge. Unfortunately, things don't always go smoothly. If they don't, you may need to make possibly far-reaching changes to the sources. Part 1 doesn't pay much attention to this kind of modification—that's the topic of part 2 of this book, which *does* expect a good understanding of programming:

- In Chapter 11, *Hardware dependencies*, we'll look at problems caused by differences in the underlying hardware platform.
- In the following five chapters, we'll look at some of the differences in different UNIX flavours. First we'll look at a number of smaller differences in Chapter 12, *Kernel dependencies*, then we'll look at some of the more common problem areas in Chapter 13, *Signals*, Chapter 14, *File systems*, Chapter 15, *Terminal drivers*, and Chapter 16, *Time-keeping*.
- We'll look at the surprising number of headaches caused by header files in Chapter 17, *Header files*, and at system library functionality in Chapter 18, *Function libraries*.
- We'll examine the differences between various flavours of the more important tools in Chapter 19, *Make*, Chapter 20, *Compilers*, and Chapter 21, *Object files and friends*.

Finally, there are a number of appendixes:

- Appendix A, *Comparative reference to UNIX data types*, describes the plethora of data types that have developed since the advent of ANSI C.
- Appendix B, *Compiler flags*, gives you a comparative reference to the compiler flags of many common systems.
- Appendix C, *Assembler directives and flags*, gives you a comparative reference to assembler directives and flags.
- Appendix D, *Linker flags*, gives you a comparative reference to linker flags.
- Appendix E, *Where to get sources*, gives you information on where to find useful source files, including a number of the packages we discuss in this book.

* With the exception of Taylor *uucp*, BSD/OS, which at the time was called BSD/386, is supplied with all these packages, so you would only be need to port them if you wanted to modify them or port a new version.

Preparations

You don't need much to port most packages. Normally everything you need—a C compiler, a C library, *make* and some standard tools—should be available on your system. If you have a system that doesn't include some of these tools, such as a System V release where every individual program seems to cost extra, or if the tools are so out-of-date that they are almost useless, such as XENIX, you may have problems.

If your tools are less than adequate, you should consider using the products of the Free Software Foundation. In particular, the GNU C compiler *gcc* is better than many proprietary compilers, and is the standard compiler of the Open Software Foundation. You can get many packages directly from the Internet or on CD-ROM. If you are going to be doing any serious porting, I recommend that you get at least the GNU software packages, 4.4BSD Lite, and TEX, preferably on CD-ROM. In particular, the GNU software and 4.4BSD Lite contain the sources to many library functions that may be missing from your system. In addition, many of the GNU packages are available in precompiled binary form from a number of sources. I'll refer to these packages frequently in the text.