Porting Berkeley UNIX® through the GNU C Compiler

John Gilmore

Grasshopper Group
San Francisco, CA, USA 94117
gnu at toad.com

ABSTRACT

We have ported UC Berkeley's latest UNIX sources through the GNU C Compiler, a free draft-ANSI compatible compiler written by Richard Stallman and available from the Free Software Foundation. In the process, we made Berkeley UNIX more compatible with the draft ANSI C standard, and tested the GNU C Compiler for its full production release. We describe the impact of various ANSI C changes on the Berkeley UNIX sources, the kinds of non-portable code that the conversion uncovered, and how we fixed them. We also briefly explore some limitations in the tools used to build a UNIX system.

Introduction

The GNU C Compiler (GCC) is a complete C compiler, compatible with the draft ANSI standard, and available in source from the Free Software Foundation (FSF). It was written by Richard Stallman in 1986 and 1987, and is (at this writing) in its 18th release. It is a major component of the GNU (''GNU's Not UNIX'') project, whose aim is to build a complete UNIX-like software system, available in source to anyone who wants it. The compiler produces good code — better than most commercial compilers — and has been ported to the Vax, MC680X0, and NS32XXX.

Berkeley UNIX, from the Computer Systems Research Group (CSRG) at the University of California at Berkeley, had its start in the 1970's with a prerelease UNIX Version 7, and has been improving ever since. The current sources derive from the 1978 AT&T ''32V'' release, a V7 variant for the Vax. CSRG has produced four major releases for the Vax — 3, 4.1, 4.2, and 4.3BSD. These releases have set the standard for high powered UNIX systems for many years, and continue to offer an improved alternative to the flat-tasting AT&T UNIX releases.

However, Berkeley's C compiler is based on an old version of PCC, the Portable C Compiler from AT&T. There was little chance that anyone would provide ANSI C language extensions in this compiler, or do significant work on optimizing the generated code. By merging the GNU C compiler into the Berkeley release, we provided these new features to Berkeley Unix users at a low cost, while offering the GNU project an important test case for GNU C.

Goals

The major goal for the project is to move GCC out of ''beta test'' and into ''production'' status, by demonstrating that a successful UNIX port can be based on it.

We are also providing a better maintained compiler for Berkeley UNIX. GCC already produces better object code then the previous compiler, has a more modern internal structure, and supports useful features such as function prototype declarations. It is also maintained by a large collection of people around the world, who contribute their fixes and enhancements to the master sources. Regular releases by the Free Software Foundation encourage distribution of the improvements. In contrast, PCC is proprietary to AT&T, and few fixes are widely distributed, except as part of infrequent and expensive AT&T releases.

We are producing a UNIX source tree which can be compiled by *both* the old and the new compilers. This is partly for convenience during the port, partly in case the project suffers long delays, and partly because Berkeley UNIX also runs on the Tahoe, a fast Vax-like machine built by Computer Consoles, which GCC does not yet support. We are avoiding the introduction of new **#ifdef**'s, instead rewriting the code so that it does not depend on the features of either compiler.

We have to constantly remind ourselves to minimize the changes required. It's too easy to get lost in a maze of twisty UNIX code, all desperately needing improvement.

Whenever we have to make a change, we have moved in the direction of ANSI C and POSIX compatability.

People

The project was conceived by John Gilmore, and endorsed by Keith Bostic and Mike Karels of CSRG, and Richard Stallman of FSF. John did the major grunt work and provided fixes to the UNIX code. Keith and Mike provided machine resources, collaborated on major decisions, and arbitrated the style and content of the changes to UNIX. Richard provided quick turnaround on compiler bug fixes and problem solving. This setup worked extremely well.

We started work on 17 December 1987, and are not yet done at the time of writing (19 February 1988). About 9 days of my time, 2 of Keith's, half a day of Mike's, and XXX days of Richard's have gone into the project so far.

Working Style

Most of the work was done over networks, in a loosely coordinated style which was hard to concieve of only a few years ago.† John worked in San Francisco, Keith in Berkeley, and Richard in Cambridge. Keith set up an account and a copy of the source tree on *vangogh*, a Vax 8600 at Berkeley. John spent a few days in front of a Sun at Berkeley getting things straight, but did most of the work by dialing in at 2400 baud from his office in San Francisco. When we modified UNIX source files, Keith checked the changes and merged them back into the master UNIX sources on another machine at Berkeley. When we found an apparent bug in GCC, we isolated a small excerpt or test program to demonstrate the bug, and forwarded it to Richard by Internet electronic mail. Bug fixes came back as new GCC releases, which were FTP'd over the Internet from MIT. Ongoing status reports, discussions, and scheduling were done by*uucp* and Internet electronic mail.

At this writing, we have used four GCC releases (1.15 through 1.18). For each GCC release, we did a ‘‘pass’’ over the UNIX source tree; one such pass included an updated source tree as well. Each GCC release was built, tested, and installed on *vangogh* without trouble. Then we ran *make clean; make* on the source tree, and examined 500K to 800K of resulting output. Keith Bostic's Makefiles did an excellent job of automating this process, though we ran into some problems with the UNIX compilation model in general, and limitations in *make* in particular.

ANSI Language Changes

The problems encountered during the port fell into two general categories. Some of the code was not written portably and failed in the new environment. Other code was written portably for its time, but failed because ANSI C has redefined parts of the language. In some cases it was hard to tell the difference; the consensus on what is ‘‘portable code’’ changes over time, and on some points there is no agreement.

The major ANSI C problem was the generation of **character constants in cpp**. The traditional UNIX C preprocessor (*cpp*), written by John F. Reiser, would substitute a macro's parameters into like-named substrings even inside single or double quotes in the macro definition. For example:

† Much of the free software work that is happening these days occurs in this manner, and I would like to publicly thank the original DARPA pioneers who gave birth to this vision of wide area, computer mediated collaborative work.

```
#define    CTRL(c)    ('c'&037)
#define    CEOF       CTRL(d)
```

In an attempt to make things easier for tokenizing preprocessors, ANSI C has changed the rules here, and there is in fact *no* way to generate a character constant containing a macro argument. (There is a way to generate a character *string*, e.g. double-quoted string, but not a single-quoted character. We consider this a bug in ANSI C.) Fixing this required altering both the macro definition and each reference to the macro:

```
#define    CTRL(c)    (c&037)
#define    CEOF       CTRL('d')
```

This required changes in about 10 system include files and in about 45 source modules. Many user programs turned out to depend on the undocumented **CTRL** macro, defined in **<sys/ttychars.h>**, and since all its callers had to change, all those programs did too.

Another *cpp* problem involved **token concatenation**. No formal facilities were provided for this in the old *cpp*, but many users discovered that with code like this, from the /etc/passwd scanning code:

```
#define    EXPAND(e)       passwd.pw_/**/e = tp; while (*tp++ = *cp++);
        EXPAND(name);
        EXPAND(passwd);
```

they could cause a macro argument to be concatenated with another argument, or with preexisting text, to make a single name. In one case (*phantasia*), the Makefile provided half of a quoted string as a command line **#define**, and the source text provided the other half! ANSI C does not allow a preprocessor to concatenate tokens in these ways, instead providing a newly invented **##** operator, and new rules requiring the compiler to concatenate adjacent character strings. Again, it was impossible to write a macro that works with both old and new compilers, and we didn't want to uglify our code with **#ifdef __STDC__**; our solution was to rewrite both the macros and all their callers, to avoid ever having to concatenate tokens:

```
#define    EXPAND(e)       passwd.e = tp; while (*tp++ = *cp++);
        EXPAND(pw_name);
        EXPAND(pw_passwd);
```

Mostly the token concatenation was used as a typing convenience, so this was not a problem. It involved changes to five modules. We found no clean solution for *phantasia*; a fix will probably involve rewriting it to do explicit string concatenations at runtime.

Changes to the **scope of externals** provided another set of widely scattered changes. If an external identifier is declared from inside a function, PCC causes that declaration to be visible to the entire remaining text of the source file. This also applies to functions which are implicitly declared when they first appear in an expression. This behaviour was not explicitly sanctioned by K&R, but it was condoned (pg. 206, 2nd paragraph), and many programs depended on it. ANSI C changed the scope rules to be more consistent; if you declare an external identifier in a local block, the declaration has no effect outside the block. We moved extern declarations to global scope, or added global function declarations, in 38 files to handle this.

A number of programs used **new keywords** such as *signed* or *const* as identifiers. We renamed the identifiers in 9 modules.

The Fortran libraries used a **typedef name as a formal parameter** to a set of functions. ANSI C has disallowed this, since it complicates the parsing of the new prototype-style function declarations. We renamed the parameter in 8 modules.

Three modules used a **typedef with modifiers**, e.g.:

```
typedef int CONSZ;
x = (unsigned CONSZ) y;
```

This has been repudiated by ANSI C. We fixed it by making the original typedef **unsigned** where possible, or by creating a second typedef for “U_CONSZ”.

Non-Portable Constructs

The worst non-portable construct we found in the UNIX sources was the use of **pointers to non-members**. There was plenty of code as bad as:

```
int *foo;
foo->memb = 5
if (foo->humbug >= -1) bah();
```

and, in many cases, *memb* and *humbug* are not even members of the same struct! Such code seems to have been written with a ‘‘BCPL’’ mentality, assuming that all pointers are really the same thing and it doesn’t matter what their type is. Early C implementations lacked the **union** declarator, and did not distinguish between the members of different structures. Exploiting this has been considered bad practice for years, and lint checks for it, though many UNIX compilers do not. We found a lot of it in old code, though newer code did not lack for examples either. Fixing this problem caused the most work, because we had to figure out what each untyped or mistyped pointer was *really* being used for, then fix its type, and whatever references to it were inconsistent with that type. We changed 5 modules due to this. One program, *efl*, would have required so much work that we abandoned it, since we could not find anyone using it.

Another problem was caused by existing uses of **cpp on non-C sources**. Various assembler language modules were being preprocessed by *cpp*, probably because there is no standard macro assembler for UNIX. These modules are carefully arranged to avoid confusing the old *cpp*; for example, assembler language comments are introduced by **#**, but indented so that *cpp* will not treat them as control lines. ANSI *cpp*’s handle white space on both sides of the ‘‘#’’, so indentation no longer hides these comments. Also, the ANSI rules to require the preprocessor to keep track of which material is inside single and double quotes and which is outside; the old *cpp* terminated a character string or constant at the next unescaped newline. Vax assembler language uses unmatched quotes when specifying single ASCII characters, such as in immediate operands. This causes an ANSI *cpp* to stop processing # directives at that point, until it finds another unmatched quote. We chose to alter the assembler modules to avoid stumbling over these features in ANSI C preprocessors, without fixing the larger problem of using a C-specific preprocessor on non-C text.

In addition to embedded C preprocessor statements in assembler sources, we had to deal with **asm() constructs** in C source. Some system-dependent routines were written in C with intermixed assembler code, producing a mess when compiled with anything but the original compiler. Other routines, such as *compress*, drop in an **asm()** here or there as an optimization. Still more modules, including the kernel, run a *sed* script over the assembler code generated by the C compiler, before assembling and linking it. There is no general solution to these problems. GCC has added an asm() facility that is independent of the compiler’s register allocation strategy, but programs using this are incompatible with the old C compiler. We are investigating a possible fix involving changing all these places to use e.g. **#include <machine/inline.h>** which, in GCC, would define inline code containing asm()s, while in PCC, declarations of (slower) external functions would be generated.

Troff used **multi-character constants** in its font tables; we fixed it with a macro for building an int out of two characters. A Fortran library module used the character constant **’EOF’**, presumably a typo for **EOF**; and *rogue* defined the character ’300’ as a possible command letter. While ANSI C permits multiple character constants, they are implementation defined, and GCC wisely defines them to be invalid (as the standard should have done).

Some programs tried to declare functions or variables, **omitting both type and storage class**. This usage is not even valid in K&R, though PCC accepts it. We fixed this in about 15 modules, by adding ‘‘int’’ to the declarations. There were two other modules where this check uncovered inadvertent use of ‘‘;’’ in a declaration list where ‘‘,’’ was intended.

GCC provides better error checking in a few ways, and caught a number of bugs caused by misunderstood **sign extension**. It warns ‘‘comparison is always 0 due to limited range of data type’’ for constructs like:

```
char c;
if (c == 0x80)  foo();
```

If a signed character contains the bit pattern 0x80, using it in an expression causes it to be sign-extended to

0xFFFFFF80, which does not equal 0x00000080. Bugs of this sort were fixed, typically by casting the 0x80 to (char), in 5 modules.

Changes to the rules for **parsing declarations** made us fix two modules where the last declaration in a struct was immediately followed by a closing brace, without a semicolon. Three more modules needed changes because the rules for where braces are required in struct or array initializers have changed. Four programs defined a **struct foo** and then referenced it as a **union foo**, or vice verse. Two programs declared **register struct foo bar;** and then took bar's address, which is not allowed for register variables!

Thirteen programs had miscellaneous **pointer usage bugs** fixed. Two more were comparing pointers to **-1**; these were changed to use zero as a flag value instead.

In ANSI C, local variables in use at a **setjmp()** are no longer guaranteed to be preserved when a **longjmp()** occurs, unless they are declared **volatile**. This is not a problem for the Vax port, since the Vax longjmp() will continue to restore the registers, but gcc warns about this situation, since code that assumes restoration is not portable. We have not yet worked on fixes for this.

Five or ten other miscellaneous bugs were caught and fixed.

Least portable UNIX code

The process of porting software inevitably uncovers a few files that cause a disproportionate share of problems. For our port, the clear winner is *efl*, the Extended Fortran Language, by Stu Feldman. It defines ‘‘**typedef int * ptr;**’’ in a header file, and then uses a ‘‘ptr’’ to point to anything. GCC produced 1600 lines of errors messages on this program alone, and three modules of it caused compiler core dumps. We ended up deciding to abandon support for it rather than attempt to clean it up.

A runner-up is *pcc*, the Portable C Compiler itself, by Steven C. Johnson. It caused GCC to core-dump twice, tickled another GCC parsing bug, and contained the modified typedef and sign extension problems mentioned above.

Third place goes to *monop*, the Monopoly† game, by Ken Arnold. This program used a variety of typed pointers, but the main pointer to a set of structs was declared as a **char ***. Another part of the code initialized an array of struct pointers with integer values, then a small loop at the beginning of the game would read out these integers and replace them with corresponding ‘‘real’’ struct pointers. It took about two days to face up to the job and about a day to clean it up.

Honorable mention for silly mistakes goes to the *indent* program, by someone at the University of Illinois. It contain the only instance of**a + = b** (with a space between + and =), and w as the only module to terminate its **#include** directives with a semicolon. It also contained a comparison between a character and the value 0200, a value that a signed 8-bit char can never hold.

Results

We are pleased with the results so far. Most of the UNIX code compiled without problems, and the parts which we have executed are free from code generation bugs. The worst of the ANSI C changes only required roughly fifty modules to be changed, and there were only two problems of this magnitude. A total of twenty bugs in gcc were located so far, and most of them are now fixed. We expected several times this many bugs; the compiler is in better shape than any of us expected.

Many minor type problems and ‘‘nit’’ incompatabilities with ANSI C have been removed from the UNIX sources.

Future Results

(This section will move to **Results** *for the final paper.)*

We expect that the size of the UNIX binaries will be significantly less than with the previous compiler, but at the current stage of the project we can't easily confirm the expectation.

When the system compiled with GCC is in everyday use at Berkeley, GCC will be relabeled as a full production-quality compiler, which will encourage its wider use.

† Trademark of Parker Brothers

Non-Results

We have not attempted to make Berkeley UNIX fully ANSI C compliant. In particular, we have retained preprocessor comments (#endif FOO) as well as machine-specific **#define**'s (#ifdef vax). GCC supports these features without trouble, even though ANSI C does not.

The UNIX kernel has not yet been ported to gcc. Other people are working on this, compiling one module at a time and running it for a while before moving on to the next. We will merge their work with ours once we have the rest of the system in a stable state.

Pieces of the Portable C Compiler are still being used inside *lint*, *f77*, and *pc*. Eventually someone will write Fortran and Pascal front-ends for gcc; this has already been done for C++. So far nobody has created a GNU *lint*, but it is an obvious project.

CSRG has ported Berkeley UNIX to the Tahoe, a fast Vax-like machine built by Computer Consoles and resold by Harris and others. We are looking for someone to do a Tahoe port of gcc, to replace the PCC supplied by CCI.

Problems in Building UNIX

UNIX compilers traditionally look in certain global places in the file system for their libraries, include files, etc. This is a problem when cross-compiling, or when building a new UNIX release (which almost amounts to the same thing). While it is possible to provide a new default directory for **#include** files, if a source program **#include**s a file that is not in the cross-compilation include files, the C compiler will erroneously use the one from /usr/include. There should be a switch that turns off *all* the built-in include file and library pathnames, and only uses those specified on the compiler's command line.

However, there is still the problem of getting those switches to the compiler's command line. *Mak e* is a great tool for dealing with one directory's worth of files, but as UNIX has evolved, *make* has not kept up. Indeed, it has fallen behind; Makefiles that worked perfectly well five years ago will no longer work because each manufacturer (AT&T especially) has hacked up their *make* to include harmful, gratuitous, and mutually incompatible changes. The result is that a Makefile that works on your system is unlikely to work on your neighbor's system, unless they are from the same manufacturer, and you happen to use the same login shell.

Make works poorly on nested directory structures, too. As an example, we could find no way to change ‘‘cc’’ to ‘‘gcc’’ in all the Makefiles used to build Berkeley UNIX (short of text-editing them all). In a single directory, you can say *make CC=gcc*, but this change is not propagated to subdirectories. You can manually propagate that change one level by saying *make CC=gcc MFLAGS='CC=gcc'* but that only goes one level (at least in Berkeley's version of *make*). We ended up putting a copy of gcc in a private *bin* directory, named *cc*, and putting that directory on the front of the search path. (When we later wanted to override CFLAGS as well, *~/bin/cc* became a shell script that invokes *gcc -W*).

Another problem with *make* is that even if it was instructed to ignore errors (with -i or -k), it exits if it can't locate a file that something else depends upon. This has the effect of ‘‘pruning’’ a potentially large section of the source hierarchy, and the only warning is an unobtrusive message buried among 500K of other output.

Of course, if someone was to fix these bugs in *make*, they would be creating yet another incompatible version. I have been watching the papers on the ‘‘new makes’’ and so far there doesn't seem to be one that handles deeply nested source trees in a clean and consistent fashion, or is otherwise so much better than *make* that it's worth the effort to switch. I think it is time to look for a completely new paradigm for software compilation control. I don't have any major insights on where to go from here, but it is clear to me that *make* and its derivatives have reached their useful limits.

Availability

These changes will be available to recipients of Berkeley's next software distribution, whenever that is. We will also make diffs available to others involved in porting UNIX to ANSI C. We suspect that most of the problems we solved have already been handled in one or another UNIX port, but the work had to be duplicated because either it was not sent back to Berkeley or AT&T, or the changes were not accepted.

(AT&T has a history of pretending that UNIX bugs do not exist, and Berkeley has limited manpower).

Future Work

Future projects include building a complete set of ANSI C and POSIX compatible include files and libraries (including function prototypes), and converting the existing sources to use them. An eventual goal is to produce a fully standard-conforming UNIX system — not only in the interface provided to users, but with sources which will compile and run on any standard-conforming compiler and libraries.

The success of this collaboration between GNU and CSRG has encouraged further cooperation. Both parties feel that AT&T licensing is a problem; most recipients of CSRG releases have old UNIX licenses, and are unwilling to upgrade to more expensive and more onerous AT&T licenses. However, new AT&T releases include some features which would be useful in Berkeley UNIX. The GNU project is working to provide early reimplementations of these features, such as improved shells and ‘‘make’’ commands. In return, CSRG is working to release software to the public which has previously been held to be ‘‘ UNIX licensed’’ even though it was not derived from AT&T code, such as the implementation of TCP/IP, and many of the Berkeley utility programs.

References

Draft Proposed American National Standard — Programming Language C, ANSI X3.J11, draft of October 1, 1986 (update for new draft when out). CBEMA, 311 First Street NW #1500, Washington DC 20001.

4.3BSD Manual Set, Computer Systems Research Group, University of California at Berkeley.

Fowler, Glenn S., ‘‘The Fourth Generation Make’’, Usenix conference proceedings, Summer 1985, page 159. (More references on ‘‘make’’ are provided in this paper.)

Hume, Andrew, ‘‘Mk: a successor to make’’, Usenix conference proceedings, Summer 1987, page 445.

Kernighan, Brian W. and Ritchie, Dennis M., ‘‘*The C Programming Language*’’, Prentice-Hall, 1978.