C and the AT&T Unix Port—A Personal History

Stephen C Johnson
Transmeta Corp

Porting Unix from the 16-bit PDP-11 to the 32-bit Interdata 8/32 had a profound effect on the C language and the way it was used in Unix.

Background

I got involved in compilers by a backdoor route— Dennis Ritchie had written a compiler for a language called B, a typeless language that was very effective on word-addressed machines. He had originally written this compiler on the Honeywell 645, a 36-bit word-addressed machine. Everything was a word—integers, characters, pointers, functions, arrays (the latter two were pointers to the code and the data, respectively). Ambiguity was broken by the operator type—if you applied unary '*' to something, it had better be a pointer. If you applied binary '*' to something, it had better not be. It was up to the programmer to decide. There were no structures and no type checking.

The ability to control memory usage was crucial in those days—our mainframes allowed only 128K (9-bit) bytes for each user program, and that included code and data. System call interfaces, which were considerable, also had to fit in that space. This was a multi-million dollar machine, one of a couple that did all the central computing at the Murray Hill Bell Labs location.

When Dennis started spending all his time with the new PDP-11, the Honeywell compiler became an orphan. I was writing some system code (a dynamic loader) and was seriously hampered by some missing operators, notably XOR. There were also a couple of bugs. Dennis has a way of saying very profound things in a very simple way. One day he urged me to fix the compiler myself. "After all," he said, "it's just a program".

This doesn't sound too profound now, but at the time most compilers (and operating systems) were written in assembler—the compilers were hard-wired into the operating system somewhat the way device drivers are now. Once I appreciated that the compiler was just a program, and could be understood by ordinary mortals, I found myself hooked into what has been a 25-year career.

Yacc is Born

Dennis' compiler was a superb introduction to the craft—both the statement parsing and the code generation were elegant and powerful. However, the expression parsing was a horrible kluge. One of the things I wanted to do was to to add an XOR operator, but my first several tries ended in failure. The method, operator precedence, required all sorts of kluges for function calls and parentheses, and adding a new precedence level in the middle required rewriting a major part of the expression parser. I was complaining about this after my third or fourth failure, when Al Aho suggested that there were some new methods recently published that would allow parsers to be generated from BNF. Al introduced me to LR parsing, and within a few months Yacc was born. It was written on the Honeywell, in B, and its early history is worth a paper in itself.

But by late 1972, Yacc had been ported to the new Unix system—there was a B interpreter, and later a compiler, for it. Yacc was a notorious processor hog, taking 20 minutes to process a 50-statement grammar on the PDP-11 ("Oh crap, Johnson's running Yacc again" people would say when their terminal stopped responding).

The PDP-11 at that time had 32K bytes of memory—the system got 16K and the users got 16K. The B compiler also ran in 16K. An

amusing incident shows just how close to the wall we were. One day Yacc did not compile—it had compiled just fine the previous day. It turns out that Dennis had added the 'for' statement, in essentially the form we know it today, to B. The word 'for' took up a symbol table slot, and, unknown to me, Yacc had been filling every symbol table slot when it compiled. So when the 'for' statement came in, I had to rewrite a routine in Yacc to use one fewer local variable.

Around this time, we got another memory board for the PDP-11. I recall it being 16K bytes, which once again was split between the OS and the users, and costing in the neighborhood of $20,000 (1972 dollars). It was hard to say who was happier, the OS writers or the users, at going from 16K to an entire 24K.

In early 1973, I went on a 9-month sabbatical to the University of Waterloo. When I returned, B was dead—long live C. An MIT co-op student named Al Snyder had rewritten Yacc from B to C, and used it as the basis for a C compiler that would run on both the Honeywell and the PDP-11. It had a Yacc grammar for the language. Neither implementation was especially compelling—it took four passes to run on the PDP-11, so it was horribly slow, and the code generator on the Honeywell had numerous bugs. But it was a very impressive few months work.

When I returned from Canada, it was strongly suggested that I take this over—Al was returning to school. Over the course of the next six months, I replaced nearly everything but portions of the grammar, and it became the first Portable C compiler. The number of passes on the PDP-11 was reduced to two, and the code quality on the Honeywell got good enough that some people in the computation center started to use it to write system code. Working with Tom Petersen, we got a version going for the IBM-370, and also for a Bell Labs switching machine called the 3A.

Over the next year or two, I was able to get the front end to be relatively machine independent—I parameterized some things like byte order and the widths of integers, as well as some code like array initialization that had to be done in the front end due to lack of space in the compiler. As I made the front ends more similar and more machine independent, I wanted a way to test that the front end could really 'understand' C. I realized that I could write a simple back end that would just make a table of where the functions were defined, with the file and line numbers. Before long, I realized that I could also find uses, and, with some simple sorting and a back-end table, I could check number and type of function arguments—lint was born.

The back end code generation was quite different. It took quite a while to bring the code generators under one umbrella, based on a template rewriting model and strongly influenced by the Sethi-Ullman method that Dennis had used in the Honeywell B compiler. The Portable C Compiler (PCC) began to appear in a useful form.

To Port a System

Then Dennis said another one of those simple, profound things, "Given the complexity of some of the applications in the Bell System, it would be easier to port the operating system to a new machine than to port the application to a new operating system." The existing operating systems at the time were mostly still punched card based (or based on card images copied to tape or disc), mass storage organization and command line syntax was completely unique for each system, interactive and background jobs had different system calls to accomplish the same tasks, there was effectively no networking, and word-sizes of 16, 24, 32, 36, 60, and 64 were in use in the Bell System (not to mention several different byte orders). He didn't actually say "Unix is just a program", but I got the idea. He asked me if I could help evaluate machines for a 32-bit port of Unix, and build the compiler for the resulting port, and I enthusiastically agreed.

There was an organization at Bell Labs that very much wanted us to use an IBM 360—they offered to buy us one if we would port Unix to it. I hated compiling for that machine, and Dennis didn't like the I/O structure, so we declined. The Dec 10 was another machine we investigated, but it was 36 bits wide—Dennis said he would consider it if we could get 10-

track tape drives for backup. At the time, the Dec 10 was the dominant University research machine, and we worried that we would cut ourselves off from universities by not using that machine. We settled on the Interdata because it was 32 bits, resembled a 360, but had a much nicer instruction set, easier to compile for.

The Impact on C

Dennis and Ken set to work on the OS, to make it truly portable. It was necessary to clean up C quite a bit in order to write the OS. In the evolution from B to C, a lot of old B code had been carried over unchanged. This meant that the uses of types were very fast and loose, or missing altogether. Before C had the unsigned data type, people would assign values to character pointers to get unsigned arithmetic, for example.

Structures had been in the language for several years, almost from the beginning of C, but all structure members were in the same namespace. By the time we started the port, there were structures heavily used in the operating system and the interfaces to it, but it was still common to see people doing I/O by writing things like:

```
* 0177320->enable = 1;
```

As long as you had exactly one structure with the member 'enable', this was unambiguous. But clearly the handwriting was on the wall—the number of names a programmer had to avoid in structure members was into the hundreds.

The solution was to adopt the semantics we have now—each structure has a separate namespace, and the type of the structure or structure pointer determines the offset and type (and legality) of the reference. But, because we had all this 'legacy' code, we decided to make an interim version of the compiler that would allow references that were unique, so old code would work, but with a warning.

The symbol table handling of PCC was never especially profound, and this change brought it to its knees. Probably half of the bugs that turned up in the porting involved really nasty problems with structure members. I had sanity checks littering the code, amusing myself by inventing synonyms for 'bad' and 'illegal'. I recall Ken Thompson handing the limping compiler a structure that contained an array whose dimension was an expression that contained a sizeof of another entire structure declaration. The PCC front end rode bravely into battle but returned on its shield. I still remember the look on Ken's face as he showed me the error message and mildly asked, "What is a gummy structure?".

In order to deal with a lot of the I/O device code, we needed casts in the worst way. Unfortunately, every syntax we could come up with was appalling. Generally, I stuck by the credo that the C language was Dennis', and he made the language decisions, although a few things came up on other machines (such as the interaction of byte order and bitfields, or the whole issue of alignment of structures) and I had to try to do something sensible. But in this case, we just needed casts too badly, so I implemented what we now know. The simple stuff was easy enough to write and understand that we were hooked before we realized the full horror that lurked behind that syntax.

Bitfields had entered C, but faced profound challenges when we moved programs to machines with different word sizes. One issue was the relationship between a bitfield of size 8 bits, and a byte of the same size. I felt they should have the same semantics, and this led to the decision to allocate bitfields in the same order as the bytes were laid out in the machine. On any given machine, this was the right answer, but this made bitfields next to useless in doing communication software.

One of the most profound changes in the way we did business had to do with header files. Previously, header files for system structures had been printed in the manual. If you wanted to use an I/O device or do a system call, you copied the structure out of the manual into your source file and went at it. This clearly was a disaster when it came to a radically different machine. Even sizeof was not much help. We concluded that we needed to have a central repository of system header files, and use `#include` to access them. I believe that

typedef was invented about this time as well so we could have symbolic names for system quantities that might be represented differently on different systems.

Lint was used in a clever way to help us find all the commands that had local copies of the system structures. Lint would complain if two calls to the same routine used physically different structure definitions, even if the declarations were identical. A summer student from Princeton, Tom Lyon, did most of the grunt work of finding and fixing the system call declarations. This was especially important because older versions of Unix had returned -1 (0177777 in PDP-11-ese) when they failed, while the new system calls would typically return 0. We had to find them all.

The Port

The Interdata arrived, as a bare machine. There were some self-loading diagnostics, but we never put a bit of the manufacturer's software on the machine. We had a disc and a tape drive, but no networking. Bootstrapping involved entering a tape boot program from the console, reading a record, and then jumping to it.

The Interdata was big-endian, while the PDP-11 of course was little-endian—this was an endless source of headaches. Also, this was the first machine either Dennis or me had used that spoke hex, and we both had trouble adapting after a decade of octal. I had to do I/O character by character to the console (it typed on real paper, youngsters!). In a day or so, I had putc working, then wrote puts. Then I started running compiler tests. In each case, I had to write a tape on the PDP-11, dismount it, carry it to the Interdata, mount it, copy it in to memory, and execute it.

There was one amusing incident. Just when it seemed things were really starting to work, I tried a somewhat bigger test, and it failed. I spent a couple of days single stepping the program, and finally concluded that we had a bad location in memory—no matter what I stored into it, it always returned 0. To make a long story short, I had been loading my programs at location 0, and had finally loaded a long-enough program that I ventured into the region of the memory address space where the relocation hardware was located. It turns out I had been executing, and even single stepping, through relocation registers just fine, but the page-fault status register was read-only, and that was the location that always appeared to be 0.

Within two weeks, I had a stable cross compiler from the PDP-11 to the Interdata working, and we could begin the job of porting. Ken would produce the kernel and get it working on the PDP-11, Dennis did most of the porting work (especially the device drivers and getting the shell to work), and I helped with the debugging.

The tape drive continued to be a major bottleneck. Because of the byte swapping, Dennis had to create a proto file image on the PDP-11 that could be copied to tape on the PDP-11, copied from tape to the Interdata, then copied to the Interdata disc. Depending on the datatype and the alignment, the data on the PDP-11 had to have the bytes swapped in one of several ways to survive this. The first thing Dennis did was to put a checksum in tar so we could tell if we had the bytes swapped. This saved time. Then he put a byte-swapper in tar so if we had the bytes wrong we could swap them as we read them. This saved more time. Then, in a stroke of brilliance, he combined the two ideas—the tape would be read, and if the checksum failed the tape would be rewound and read again with the bytes swapped. Overnight, byte swapping on the tape drive disappeared as an issue...

After a couple of months (and some hardware problems that caused the machine to be sent back to the factory for a couple of days) we were ready to dispatch to the first user program. The routines, saveu and retu, were elegant and simple on the PDP-11, but wouldn't work for anything on the Interdata. Dennis and I had many late-night phone calls about this, and finally he totally rewrote these routines and we were able to make progress. These were the notorious 'you are not expected to understand this' routines—I don't ever recall seeing that comment when I was staring at them, and it was 15 years before I saw the comment, and the routines, on a T-shirt and

realized that these were the routines I had lost so much sleep over.

As with any complex project, it was hard to say exactly when it first 'worked'. We had some lingering bugs (especially in the floating-point area). The numerical analysts in the department were very interested in the machine, because the PDP-11 address space was so small it was difficult to solve practical numerical problems.

As a hardware platform, the Interdata was rather nice—its architecture was reasonably clean. Unfortunately, the implementation had a couple of serious problems. None of the Interdata 'native' software did paging, and there were some failures in this area that compromised Unix stability. The one that was most painful was doing a word indirection through -1 (the old failure return, remember...). This caused both a memory protection fault and an alignment fault, which curdled the microcode so badly the system could not recover—it actually made the next several instructions execute incorrectly. Since this happened even in user mode, we needed to have this fixed.

At a high-level meeting that included Dennis and me, many of our bosses, and many bigwigs from Interdata, we painted a glowing picture of how many machines they could sell to the Bell System if they would just fix a few small problems. They thought about it and said no. Six months later, the VAX was announced and the rest was history. . . .