AUUGN

July 1982

Vol IV No IV

Australian UNIX Users Group

NEWSLETTER

Editorial.....1

How to Port UNIX to a Microcomputer.....2

F77 Performance.....8

Sign Extension and Portability in C.....15

Dear Abby.....24

Netmail.....26

L A S T   I S S U E   O F   A U U G N !

I hope that got your attention because that is what you will see on an issue of AUUGN in the near future if we don't get more contributions. The most difficult apsect of producing AUUGN is not the physical production, printing etc but getting sufficient articles to print. There has been some criticism that many articles recently have been reprints from other journals or Unix conference proceedings. Other articles tend to come from people at the University of Sydney or the University of NSW, this is because these people are nearby and can be badgered into producing something. If you don't like this situation then only YOU can change it.

A short, one page article describing your current project would be a start. Alternatively you could describe your system, what changes/improvements you have made, the configuration and what your users do with it.

I would also like to propose that we have special issues of AUUGN devoted to particular topics. For example a special issue on teaching with Unix systems would be of interest to many university users. Special issues could be produced on graphics, real time applications, commercial software or any other topic that would be of interest to Unix users. Potential guest editors should contact me if they are interested.

If you really can't think of an article to write then we would still like to hear from you. Your ideas on what AUUGN should contain would be most appreciated.

Bob Kummerfeld

How to Port UNIX to a Microcomputer

Alan Whitney

Microsoft, Inc.
XENIX Group*
Bellevue, WA 98004

ABSTRACT

This paper will describe the mechanics of the process Microsoft has used to port UNIX to various microprocessor systems. At the end of 1980, our group accepted the challenge of porting UNIX to several microprocessor systems on a tight schedule. This paper describes some of the tools and techniques that we used to accomplish that goal.

The minimum hardware and software requirements for a target machine and the hardware configuration for our host machine are described. The process of turning a tested kernel into a marketable product is also discussed.

Introduction

At the end of 1980, our group was faced with the prospect of porting UNIX to three microprocessors: the Z8000, the 8086 and the 68000. We felt that it was absolutely necessary to develop some basic strategies that would allow us to efficiently debug the system, even in light of hardware problems and deficiencies.

Hardware configuration

I will start by describing the configuration of our development machine. We have a PDP 11/70 with 1 Mbyte of memory, two 300 Mbyte disks and three Able DHs. For each of our port targets, we use two of the DH lines; one for downloading and the other for a terminal on the 11/70.

A hardware configuration suitable for porting XENIX includes a 16 bit microprocessor with some memory management

*XENIX is a Trademark for Microsoft's version of UNIX.

scheme, at least 256 Kbytes of memory, and at least two serial interface lines. A hard disk and controller is required at some point, but it is not needed early in the porting process. Some removable storage medium is also helpful, but is not required. An 8086 based system also requires additional hardware to implement a system/user state and reserved instruction traps.

Target machine software support

In addition to the minimum hardware described above, the target must have a ROM monitor capable of changing and displaying memory and loading programs over the serial line. A breakpointing capability and subroutines that will print data to the console are also helpful. A useful alternative to the breakpointing capability is a hardware device that controls the CPU, such as an in-circuit emulator.

Debugging tools

To aid the porting process, we have developed several software tools. For the debugging phase of the kernel, we have implemented a protocol for using the second serial line on the target as a mass storage medium. This technique, which we call a "simulated disk," allows us to proceed with the port before the disk hardware has been delivered. We have created a block type disk driver which talks to our 11/70 over the serial line, requesting disk blocks to be read or written. A program on the 11/70 then performs the read or write on a file and in the case of a read, sends the data back over the serial line.

The simulated disk is used as the root file system early in the port process. An option to the program on the 11/70 allows us to display the disk blocks being accessed. The transactions are also recorded in a log file. This allows us to understand what is happening with the file system without putting any instrumentation into the system itself. When the kernel is healthy enough to reside on the hard disk and the disk driver has been tested, we copy the simulated file system to the hard disk and start running with the root there. At this point we start using the serial line as a "simulated tape" to transfer Tar format files to the target.

We have developed a subroutine called "debug" that we load with the kernel. Debug allows us to display and change memory, set breakpoints and display process status. This subroutine, together with copious debugging printouts, makes debugging the kernel straightforward.

However, these techniques are not sufficient to address timing and performance problems. In cases where hardware performance monitoring techniques are difficult or inconvenient, we have written specialized code to collect performance statistics. Unfortunately, we have found this software performance analysis to be somewhat unproductive. It generally involves writing very specialized, throw-away code. In addition, the code to collect the information typically affects the information being collected.

Program testing

To verify the operation of the system, we have written a set of test programs which will verify the operation of the system calls. We usually run a very simple program as the initialization process in the early debugging phase. This may be either a simple shell or one of the test programs described above. Once we have verified the basic operation of the system, we install the standard init and shell. In this way, we can exercise much of the system code before running any significant user programs.

We have also developed a set of shell procedures that will exercise the utility programs. At present, this set of tests is incomplete and is not exhaustive, but it does test some of the larger and more complex utilities. Bugs in the more commonly used utilities show up through casual use during the early phases of the port.

Distribution systems

To aid in tracking changes to the system software, we have developed a mechanism to save copies of all our released software. This system also allows us to keep track of changes made for our local use. The system is modeled after SCCS (Source Code Control System) at the functional level, but is oriented toward the large number of files on a distribution tape.

To allow distribution of the system for machines without magnetic tape drives, we have developed a mechanism for distributing the system on floppy disk. The distribution set created by this mechanism consists of a floppy based system diskette and several tar format diskettes containing the distribution.

How to

Up to now, I've described the environment in which the system is developed. I will now describe the sequence in which we typically attack a port.

1. Write the downloader program.

Often this task has already been done once for a similar loader format and will not have to be repeated.

2. Write a standalone program for the machine.

Our first program generally just prints some message on the console and exits. It gives us a simple test of the compiler and assembler. If this is the first port to a new processor type, some assembly code has to be written for the machine language startup. If another port to a system with the same processor has been done before, the existing assembly code has to be modified.

3. Modify the simulated disk driver and run it standalone.

This step is one of the most important in the early phase of the port. This test program is the first significant program for the machine and can turn up some serious problems. This test provides a good workout for the compiler and assembler and can point out some bugs in a new compiler. Because this program does lots of things with the serial line, the peculiarities of the serial interface are uncovered. Timing and interrupt latency problems rear their ugly heads. This is also a good test of the OEM's documentation: we try to unlock the secrets of the hardware from cryptic notes, fragments of device drivers (for example, CP/M) and more successfully, the schematics. Once we complete this step, we have become intimately acquainted with the hardware.

4. Write the memory management code.

The next step is to lock oneself into one's room for a week (two weeks for some of us, but we go home in the meantime) and write the memory management specific section of the system. This task also includes writing or modifying the bulk of mch.s; the startup, the trap handling and the low-level memory management routines. The C language section of the system is easier to port because of some things we did in our first port. The main part of the system has been partitioned into three directories; mdep for the machine dependent C code, sys for the machine independent part of the system and io for the device drivers. We have also isolated most of the memory management code into a single module which is very portable to a paging type of memory management scheme. This code is debugged up to the point at which the read of the super block takes place early in the startup code.

5. Install the simulated disk driver into the system.

Since the simulated disk driver has already been thoroughly tested, this step is a "piece of cake." At this point, we can attach the simulated disk to a file system and run the kernel up to the point of exec-ing the initialization process. The final bugs in the simulated disk driver and many of the bugs in the MMU code can be exterminated in this step.

6. Run the kernel test programs.

The next step is to run some of the kernel test programs as the initialization process. This works out bugs in the system call code, particularly problems in the MMU code having to do with the normal running of user programs. By the time this step is finished, the *fork*, *exec*, and *brk* system calls are well tested.

7. Install /etc/init and /bin/sh.

A debugging version of /*etc*/*init* and /*bin*/*sh* is installed and can be used to run more of the kernel test programs. The simulated disk driver should now be a solid, trusted piece of code. This is a good time to test the *signal* system call. Once this is done, there is no reason not to install the standard init and shell.

8. Write and debug the hard disk driver.

The next step is to debug the hard disk driver. If the disk hardware is not yet ready, this step can be postponed somewhat, but the timetable suffers if much development must be done on the simulated file system. While the simulated disk is a useful development tool, it is too slow for the debugging that follows this step.

The hard disk driver is generally developed and tested first in a standalone environment, then installed into the system for more testing. When the driver has been debugged, the simulated file system is copied to the hard disk. It is typically copied as an image and then the file system size is patched to reflect the size of the hard disk. The root and swap file systems are changed to reside on the hard disk.

9. Download utilities and test.

The next step is to download the system utilities and test them. Once confidence in the system is gained, we apply what we call the "torture test," in which we start up many processes to simulate a heavy user load. We try to run the torture test for a long period of time, e.g. over a weekend. The torture test can point up many remaining problems in the system that do not show up in casual use.

10. Port the compiler.

If this is the first port to a particular processor, we install the compiler, assembler and loader onto the machine. Also, if it has not been done already, *adb* and any other machine specific tools are modified and installed.

11. Prepare for distribution.

The next step is to prepare the system for distribution. This involves verifying that all the files to be distributed are there, that they have the correct permissions and owners, and that an effective distribution mechanism is in tact. Typically, the distribution mechanism is tested by creating a distribution set, putting up the system on another machine, and then checking that the resulting file system is similar to the one from which the the distribution set was created.

F77 Performance

David A. Mosher
Robert P. Corbett

Computer Systems Research Group
University of California
Berkeley, California 94720

April 1, 1982

ABSTRACT

The Fortran compiler under UNIX† has been the object of many heated battles. This paper describes work done to improve the performance of the Fortran compiler to a level comparable with the Fortran compiler under VMS‡ on a VAX‡.

Introduction

Fortran, though not a dominant language in our work, is still heavily used in other research efforts and in the commercial world. Many companies have invested millions of dollars in programming efforts using Fortran. In addition, there are a number of applications oriented tools (such as SPICE) used by research organizations which are written in Fortran for reasons of portability. Converting these programs to other languages would be costly and time consuming. Thus, having a Fortran compiler on a system is imperative.

The original UNIX Fortran 77 compiler was written to be portable. *F77* has been brought up on new machines in as little as two days. The performance of the compiler was considered a secondary factor in the design of the original compiler. The only real criterion was that it should produce correct code for Fortran 77 programs on a wide variety of machines.

For serious Fortran applications, the efficiency of the object programs is almost as important as correctness. A large percentage of Fortran programs are compute bound, and so the relative efficiency of codes produced by different compilers is easily noticeable. Many VAX users have expressed concern about the efficiency of the codes produced by the original *f77* compiler. Therefore, an effort was organized at Berkeley to add an optimizer to *f77*.

The remainder of this paper discusses our improvements to the *f77* compiler and our analyses of the problems yet to be solved.

† UNIX is a trademark of Bell Laboratories
‡ VAX and VMS are trademarks of Digital Equipment Corporation

Addition of a Simple Optimizer

Our initial goal was to add a basic optimizer to the compiler. To provide a framework for the addition of these optimizations, a scheme was coded to buffer incoming statements into an internal buffer of a fixed size. The fixed size was necessary to maintain compatibilty with machines which have smaller address spaces. Since there was no guarantee that enough information could be buffered to take advantage of more than a simple block of statements, the optimizer only works over blocks of statements which are known to have a single entry point.

The two initial optimizations were common subexpression elimination for basic blocks and invariant code motion for DO loops. A simple example where these optimizations take place is:

```
dimension integer a(2800,1100)
do 10 i = 1, 100
      do 20 j = 1, 200
            n = i * 5 + j * 3
            m = n * 2 + j * 3
            a(m,n) = 1
            a(0,n) = 2
         a(i,1) = 3
      20 continue
10 continue
```

This program segment includes both the implicit and explicit occurrences of the types of codes we expected to optimize. The two instances of the subexpression 'j * 3' are examples of explicit common subexpressions. The common subexpressions generated by the array references 'a(m,n)' and 'a(0,n)' are implicit. The address calculations for those references include the implicit subexpressions '2800 * n'. In both cases, the optimizer will produce code to reuse the value obtained from the first instance of each of those subexpressions in place of the code to reevaluate them. Similarly, the explicit expression 'i*5' and the implicit address calculation for 'a(i,1)' are invariant over the inner loop and both will be moved out of the loop.

Optimizing array references has proven particularly troublesome. Special consideration is needed to utilize the VAX addressing modes which support array indexing. Applying optimizations without checking for special cases results in a substantial performance degradation.

Adding those optimizations proved to be of significant benefit for most of our benchmark programs; however, the benchmark *fft* was still much slower than the version compiled using the VMS compiler. Close analysis of *fft* showed that the optimizations performed by our basic optimizer had already been carried out through careful hand optimization of the source code.

A significant deficiency of the compiler was its lack of global register allocation. Adding a simple register allocator promised a large payoff.

Allocating registers for simple variables and common subexpressions had a minimal impact on the speed of *fft*. However, after the compiler was changed so that registers could be used as base registers for arrays, its speed improved dramatically.

The basic optimizer was completed but did not achieve expected results. Analysis of the optimized code led us to believe that the differences in the execution times of the codes produced by *f77* and VMS Fortran could not be traced solely to the lack of global optimizations.

The Real Problems

By comparing the codes produced by the VMS Fortran compiler and the second pass of the portable C compiler, we discovered two major problem areas.

One problem was that double precision calculations and functions were often generated where single precision was sufficient. An error was found in the backend code generator which forced calculations following a unary operator to be done in double precision. The use of double precision math functions was viewed as a problem but we could not accurately assess the cost because of the complexity of the benchmark programs. Further analysis revealed that the conversion from single to double precision took 5 times longer than instructions which would have accomplished the same result. An integer to floating point conversion took 10 times as long as a simple move instruction. Conversion of numerical types turns out to be very expensive and should be avoided.

During our investigation of the aforementioned problem, we noticed that the VMS Fortran compiler used integer move instructions to move floating point numbers instead of the corresponding floating point instructions. At first glance, this seemed to be an ugly abuse of the instruction set. Later, we concluded that the integer move instructions could be used without ill-effect as long as the next instruction was not a conditional branch. But the question remained "why use the integer instructions?" Timing of these instructions showed that the floating point move instruction took 3 times as long to execute.

With the addition of a global register allocator, we noted that the portable C compiler required two registers be assigned to every floating point value. Further, those registers must be paired and aligned on an even register boundary. Since the VAX needs only one register to hold a single precision value, and makes no alignment requirements, those requirements are extremely wasteful. Because all floating point operations generated by the C compiler are double precision, that pattern of register allocation has been ingrained in the code generator. We are continuing to work on this problem.

The second problem was that the compiler did not generate indexing modes. In timing tests, we found that an arithmetic shift was extremely expensive and that the equivalent elementary code to replace indexing modes was significantly more expensive than the use of indexing modes. Allocating registers to serve as base registers for arrays caused some index mode instructions to be produced. However, there were still many instances in which index mode instructions should have been generated but were not. In an attempt to understand the problem, a patch was added to the code generator to produce index mode instructions where applicable. Still the problems persisted. After a careful study of the code generator, we determined that the common subexpression optimization was inhibiting the use of index mode. We also found that the trees produced by the original *f77* compiler were not of the form the portable C compiler recognized as candidates for index mode. To resolve this problem, special cases had to be introduced into the optimizer for producing subscript codes for the VAX. While implementing those special cases, we uncovered a coding error in the original compiler which was responsible for the expression trees not being in the proper form to generate indexing modes.

With relatively few changes but a much better understanding of the code generated, the code ran significantly faster. In all, one VAX dependency and a few errors accounted for much of the inefficiency of the code produced by *f77*.

Closer, But Still More to Do

At this point, the timings for some of the benchmarks were within 10% of the reported timings of VMS Fortran generated code. The major exceptions were *fft* which was a full 100% slower, and *bench2* which was 85% slower. Because the code produced for *fft* by our compiler was similar to that produced by the VMS compiler, we began to have serious doubts about the reported times given for *fft*. Our first task was to reproduce the timing results. After a few tries and a midnight session, the reported times were reproduced to within 10%.

Before our timing experiments, we did not know if the timing results produced by VMS even measured times in the same units as UNIX. Also, the methods used to obtain timing data were very different. A function was developed to return a timing value with the measurement characteristics described in the VMS manuals. This function showed that little of the time reported by timing commands on UNIX was due to startup or cleanup of a program. With this tool in hand, the task was to prove that 1 unit of time on VMS was equivalent to 1 unit of time on UNIX.

A Close Analysis of FFT

Because the execution time discrepancies were greatest for *fft*, it was used to verify the VMS timings. Solving the riddles of this program would shed a light on the remaining problems.

The *fft* program has three major sections: initialization, a fast Fourier transform, and an inverse fast Fourier transform. In measuring each section, we found that the initialization took 2.6 times longer than the measured times under VMS. Since the initialization code was small and relatively simple, running the assembly code generated by the VMS Fortran compiler under UNIX would provide a simple comparison of the timings units of VMS and UNIX.

Our approach was to start with the assembly language produced by *f77* and transform it into the assembly language produced by the VMS compiler. The first glaring problem with the code generated by *f77* was the poor handling of complex values. By recoding the complex assignment to a generatable code sequence, the initialization took 2.5 times longer.

The effects of using double precision functions for single precision calculations had to be assessed. The exponential function in the initialization code was replaced with a handwritten assembly language routine for computing a single precision exponential function with the argument and result passed as double precision values. This change made some difference but did not prove significant. The exponential function and the calls to it were changed to single precision. The initialization then took only 1.2 times longer than the code generated by the VMS Fortran compiler. Once again, unnecessary type conversions had proven to be the major bottleneck. Additional work is needed in this area to provide both single and double precision library routines and the interface to these routines.

The remaining differences in the initialization code were that the C backend code generator used the *acbl* instruction in a case where the *aobleq* instruction could have been used, a few register variables were allocated differently, and a few extra move instructions between registers and variables were being generated. When the *acbl* instructions were changed, the initialization code took 1.1 times longer. This result was a surprise since the change was so minor. Later timings showed that an *acbl* instruction takes twice as long to execute as an *aobleq*.

With the removal of extra assignments of register variables to their memory locations and adding a few more register variables, the initialization code ran as fast as the VMS Fortran generated code. We proved that the timing units were indeed the same and learned a fair amount about the VAX.

In summary, the major problem in the initialization code was the use of double precision functions where single precision routines could have been used.

The Final Analysis of FFT

At this point, *fft* took 1.9 times longer on UNIX than its VMS counterpart. There were four main classes of differences between codes produced by the VMS compiler and those produced by *f77*.

We found that VMS was able to make better use of registers. Variables in inner loops which our optimizer kept in memory were moved to registers. With this change, *fft* took 1.85 times longer than its VMS counterpart.

In our investigation, we found two interesting results. Special casing of powers of 2 in the code produced for the *power of* operation had little effect. We also found that the compiler generated elementary code for what could have been an *acbl* instruction with an increment of 2. After replacing the elementary code with an *acbl* instruction, we found performance suffered. This result was confirmed by comparisons of the *acbl* instruction with other equivalent codes.

The next problem area was the use of double precision math function. We had hoped that changing the math function calls from double to single precision would account for most of the slowness of the fft calculation since this problem was quite significant in the initialization code. But it turned out that using single precision math functions only improved the performance ratio to 1.7 times longer. We obtained slightly better performance by reducing the number of coefficients used in the cos and sin functions to be adequate for single precision use.

We were able to gain a small improvement by putting a few more variables into registers. The execution time for *fft* dropped to 1.6 times slower when single precision values were no longer spilled unnecessarily.

By eliminating six move integer from register to memory instructions, we improved performance to 1.5 times slower.

We found that the DO loop code was quite different. We adopted the VMS DO loop code and found little difference in performance. Our study of DO loop codes showed that the code *f77* produces for floating point DO loops does not conform to the Fortran standard. The standard defines the number of iterations of a DO loop to be given by an expression which is evaluated at the start of the loop. We found that the number of iterations of loops generated by *f77* need not be close to the number indicated by the value of that expression. Tests of other Fortran 77 compilers have shown that they too fail in this area. Presumably, floating point DO loops are used so infrequently, that this failing has not been noticed. The code produced by the VMS Fortran compiler is technically correct but has a number of surprising properties. In particular, the final value of the loop control variable can be far greater than the limit value.

In summary, global register allocation was a major problem area in the fft calculations. The cost of double precision math function was significant but not great.

The Remaining 50%

At this point, the assembly code running under UNIX looked almost identical to the code produced by the VMS compiler. The only remaining difference of any significance was that *f77* produced absolute addresses for static variables, while VMS used base register/displacement addresses. Because our investigations used assembly language programs, we were slow to recognize the importance of the different choices of addressing modes. Absolute mode addresses on the VAX require five bytes per address. Displacement addresses typically require only two or three bytes. Therefore, object programs produced by the VMS compiler could be as much as 50% smaller than those produced by *f77*. Because of the VAX instruction buffer, the size of the object code can have a large effect on its execution time.

F77 has now been modified to produced the same addressing modes as the VMS compiler. With this change alone, the execution time of *fft* is reduced from 24 secs. to 17 secs. When our basic optimizer is used together with the new addressing scheme, its execution time drops to 12 secs. The VMS compiler is still somewhat faster, but the difference is no longer as pronounced. Over our set of benchmark programs, the average difference in executions times is now between 10 and 15%.

Conclusions

In conclusion, the lack of a global optimizer was a key problem but not the only problem. *F77* uses double precision library functions for both single and double precision computations because the C libraries do not include double precision routines. But as shown above, the conversions added to the object code to support such a scheme are very painful on the VAX. Also, some of the instances in which bad code was produced could be traced to a few simple errors in the compiler which had major ramifications. But in the final analysis, the extra bytes needed for absolute addresses were the most significant factor in the performance.

Significantly, the speed of Fortran object programs under UNIX was found to be related to the 'maturity' of the compiler. In no way was the UNIX system itself found to adversely affect program execution speed.

Future Work

We are now in the process of producing a more elaborate global optimizer. New optimizations specifically keyed to the VAX are to be added. Also, the old buffering scheme is being replaced with a more flexible scheme which will permit more global data flow computations to be performed. We hope that the work underway will resolve the problems cited above and produce a compiler which generates code equal in execution speed to the code generated by the VMS Fortran compiler.

In addition, we are now making a major effort to improve the reliability of the compiler. We also hope to improve the facilities provided by the compiler to aid in debugging.

Acknowledgements

We would like to express our appreciation for gift funds from TRW which made this work possible. We would like to acknowledge the contibutions of the other members of the Fortran group: Channing Brown and Alastair Fyfe. We would like to thank Tom Ferrin at the University of California at San Francisco and Mike O'Dell at Lawrence Berkeley Laboratories for access to VMS systems.

We would like to acknowledge the work done by Professor Kahan for the new math library functions and Dave Wasley for the I/O library.

References

For previous discussion on Fortran performance, see *UNIX and VMS – Some Performance Comparisons* by David Kashtan, *Comments on the performance of UNIX on the VAX* by William Joy, and *Performance Issues of VMUNIX Revisited* by Thomas E. Ferrin.

Sign Extension and Portability in C

Hans Spiller

XENIX Group
Microsoft, Inc.
Bellevue, WA

ABSTRACT

The definition of the C language has left confusion regarding type casts, sign extension, and the char type. This paper discusses the Bell standard and describes non-compatible and even incorrect code in existing compilers. Due to the prevalence of such compilers, macros are presented which will convert consistently, even given an incorrect compiler. A change to the type conversion rules in The C Programming Language is proposed.

The Issue

C has 3 integral sizes: char, short, and long. These integers can, in principle, be signed or unsigned. However, early implementations did not implement unsigned, and not all current implementations implement unsigned for all sizes. Some early implementations also did not support long.

C supports both automatic and explicit conversions between any of these types. (And a few others, for that matter...) In general, converting from a longer size to a shorter is easy: just forget about the high order bits and you're done. However, in converting from a shorter to a longer type you need some way of deciding what the high order bits will be. Normally, zero extension or sign extension are used. Precisely which extension is chosen is often relevant.

```
     char c;
     int i;

1)   if (c==0200) ....

2)   i |= c;
```

In the first example - the comparison - the constant 0200 is an int, and has leading zeros on. To do the comparison, the char is sign extended to int; and should it have started out with the value 0200, it ends up with the value 0177600, and the test fails. Thus the test always fails. A good optimizing compiler could remove the test and whatever code might be executed should the test be successful.

In the second example, if the sign bit (bit 7) of the char is on, all the high order bits of the result are set regardless of their previous value. This could be a surprise to an unsuspecting user. However, if zero extension is done, the high order bits are left alone.

Modern C compilers provide a special type to do this, unsigned char, which has the property that zero extension rather than sign extension is done on instances of it. They also provide a mechanism (inline type casts) whereby you can specify unsignedness within an expression should you have the wrong type to start out with. There are unfortunately several problems.

The Problem

1) Not all C compilers support declarations or casts of all unsigned types.

2) Not all C compilers, even when they provide these types, use the same algorithm for choosing the conversions.

3) The defining document The C Programming Language, by Kernighan and Ritchie, known as the white book, explains the the standard in a confusing and almost ambiguous way. Worse, it says that whether the type char is signed or unsigned is machine dependent.

4) Given that an implementation takes advantage of the looseness of the white book and implements char as unsigned, there is no way to do sign extension of a char within the language.

If we are interested in portability, saying that something is to be machine dependent is useless. Worse yet, that the type of extension done is machine dependent is not even true. I have examples from three compilers for the PDP-11 below, and they are all different. And even though the compilers are allowed to differ where it is hard for the hardware to sign extend, most of the compilers for other machines, including Microsoft's Z8000, 8086, and 68000 compilers, all try to simulate the PDP-11 by sign extending, even when it is painful.

The reasons behind all of this are primarily historical, but the big motivating factor is trying to provide something useful. Unfortunately, before the full generality of some things had been thought out, their usefulness had gotten to the point that they were implemented.

Whats Out There

It turns out that Bell has put forward, in the form of implementations, a standard, even though many of its own compilers do not conform to it. That this Bell's intent has been confirmed by Dennis Ritchie, in a letter he wrote me last October 23. Loosely, the standard is as described in the white book, with the principle difference being that char is always signed. What the white book says about when to sign extend is in fact consistant, but the way it accomplishes this is different based upon how you read the rules. The rules are not actually ambiguous, but they are tricky. And they are by no means easy to follow. What Ritchie, the current Bell compilers, and the white book all say is that you sign extend when the operand is signed, and you zero extend only when the source is unsigned.

Not all the recent compilers from Bell implement this rule. Conveniently, there are two different families of compilers for C available on the PDP-11, one written by Dennis Ritchie of Bell Labs, which I call the Ritchie compiler, and one written by Steve Johnson, also of Bell Labs, which I call the Portable C Compiler, or PCC. I have considered two different implementations of the Ritchie compiler, one from version 7, and one from the new release 3.0. The test program is basically the same for all three, except where the Version 7 Ritchie compiler objects. For this example, the 3.0 PCC does exactly the same thing as the Version 7 PCC:

```
int i;
char c;
unsigned int ui;
unsigned char uc;
main()
{
    i = c;
    ui = c;
    i = uc;
    ui = uc;

    i = (unsigned)c;
    ui = (unsigned)c;
    i = (unsigned)uc;
    ui = (unsigned)uc;

    i = (unsigned char)c;
    ui = (unsigned char)c;
```

```
                        i = (unsigned char)uc;
                        ui = (unsigned char)uc;
                    }
```

```
      V7 PCC                  V7 Ritchie            3.0 Ritchie
**********************************************************
```

First, uncasted assignments from 8 to 16 bits.

```
  /i=c;                  /i=c;                  /i=c;
movb    _c,r0          movb    _c,r0          movb    _c,r0
mov     r0,_i          mov     r0,_i          mov     r0,_i
  /ui=c;                 /ui=c;                 /ui=c;
movb    _c,r0          movb    _c,r0          movb    _c,r0
mov     r0,_ui         mov     r0,_ui         mov     r0,_ui
  /i=uc;                                        /i=uc;
movb    _uc,_i                                clr     r0
bic     $!377,_i       /not implemented       bisb    _uc,r0
                                              mov     r0,_i
  /ui=uc;                                       /ui=uc;
movb    _uc,_ui                               clr     r0
bic     $!377,_ui                             bisb    _uc,r0
                                              mov     r0,_ui
```

Up to this point, all three compilers agree, as far as they go. The specific code generated is different, but it does the same thing. But notice that the V7 Ritchie compiler does not implement unsigned char.

Now cast to unsigned:

```
V7 PCC                 V7 Ritchie             3.0 Ritchie
**********************************************************
  /i=(unsigned)c;        /i=(unsigned)c;        /i=(unsigned)c;
movb    _c,r0          movb    _c,r0          movb    _c,r0
mov     r0,_i          bic     $-400,r0       mov     r0,_i
                       mov     r0,_i
  /ui=(unsigned)c;       /ui=(unsigned)c;       /ui=(unsigned)c;
movb    _c,r0          movb    _c,r0          movb    _c,r0
mov     r0,_ui         bic     $-400,r0       mov     r0,_ui
  /i=(unsigned)uc;       mov     r0,_ui         /i=(unsigned)uc;
movb    _uc,_i                                clr     r0
bic     $!377,_i       /not implemented       bisb    _uc,r0
                                              mov     r0,_i
  /ui=(unsigned)uc;                             /ui=(unsigned)uc;
movb    _uc,_ui                               clr     r0
bic     $!377,_ui                             bisb    _uc,r0
                                              mov     r0,_ui
```

Up to this point, the 3.0 Ritchie compiler and the PCC are in agreement. The type of extension is based on the type of the operand to the cast operator, but the cast itself has no apparent effect on the generated code, apparently because it is the same size as the destination of the assignment. The version 7 Ritchie compiler does something quite different, however. Rather than considering the type of the operand in determining the extension, the cast is used. This has the advantage that users have the opportunity of determining the extension they want, even though the full generality of the white book and 3.0 compiler is not implemented.

Now cast to unsigned char:

```
V7 PCC                 V7 Ritchie          3.0 Ritchie
***********************************************************
  /i=(unsigned char)c;  /not implemented   /i=(unsigned char)c;
movb     _c,r0                            movb     _c,r0
mov      r0,_i                            bic      $-400,r0
                                          mov      r0,_i
  /ui=(unsigned char)c;                     /ui=(unsigned char)c;
movb     _c,r0                            movb     _c,r0
mov      r0,_ui                           bic      $-400,r0
                                          mov      r0,_ui
  /i=(unsigned char)uc;                     /i=(unsigned char)uc;
movb     _uc,_i                           clr      r0
bic      $!377,_i                         bisb     _uc,r0
                                          bic      $-400,r0
                                          mov      r0,_i
  /ui=(unsigned char)uc;                    /ui=(unsigned char)uc;
movb     _uc,_ui                          clr      r0
bic      $!377,_ui                        bisb     _uc,r0
                                          bic      $-400,r0
                                          mov      r0,_ui
```

Here, the V7 Ritchie compiler does not even try, because with casts, whatever conversion is wanted can be generated, but the PCC does something which is useless. I'm fairly sure its a bug, but it completely ignores the cast. Remember that casts are supposed to be treated as though there were an assignment to a variable of the type of the cast? Where'd it go? In general, it would appear that the V7 PCC bases sign extension strictly on the type of the source operand, and ignores casts and the destination type, while the Ritchie compiler bases sign extension on the source type, unless casts are present, in which case it bases it upon the type of the cast. This means that with the PCC on the 11, it isn't possible to do unsigned extension without explicitly putting the interceeding assignment in, or by masking, should the source be in a signed type. It turns out that this problem persists when converting an int to a long in the PCC.

Despite the fact that the Ritchie compiler does not provide any way to do anything with unsigned chars, it <u>does</u> provide what is necessary: if you have an 8 bit <u>quantity</u>, you can decide with the cast operator whether you want it to be sign extended or not in a fairly convenient way. This is useful, and seems to me to be quite natural. Unfortunately, the PCC did not implement casting this way. It continues to use only the type of the source to determine the type of the cast, so there is no way to write code that will do the same thing under both version 7 compilers without explicitly masking.

The Microsoft compilers for the Z8000 and 68000 implement the same rules as the 3.0 Ritchie for these examples. The current Microsoft 8086 compiler implements the same rules as the V7 Ritchie compiler. (Little guys have these problems too...)

I have another example that I think is interesting:

```
    char c;
    long l;
    main()
    {
       l = (unsigned)c;
    }
```

```
V7 PCC                V7 Ritchie         3.0 Ritchie
*******************************************************
movb     _c,r1        movb     _c,r0     movb     _c,r0
mov      r1,_l+2.     bic      $-400,r0  mov      r0,2+_l
sxt      _l           mov      r0,2+_l   clr      _l
                      clr      _l
```

The V7 and 3.0 PCC sign extend this, the V7 Ritchie zero extends, and the 3.0 Ritchie sign extends to short, and then zero extends to long. If you use the current Bell standard - using the type of the operand of each operator to determine extension and applying it first to the signed char source, sign extending it to unsigned, and then zero extending the unsigned to signed long - then only the 3.0 Ritchie compiler produces correct code.

Dennis Ritchie, in personal correspondence, agrees that this mixed sign/zero extension is odd, but points out that what is being done is odd in itself, in that a signed quantity is being forced to fit into an unsigned hole. Extending this argument, it seems to me that as soon as you know that there is something odd going on with signedness, all the compiler can possibly know about what is happening is that there is bit pushing going on that happens to have once been based on signed numbers.

But that doesn't resolve the problem, except in the best of all possible worlds, where everybody has compilers that work consistently for a variety of machines. Bell seems to be going in that direction with release 3.0, but they are not there yet with the PCC as of 3.0, and there are all sorts of strange variant compilers in the world, including some that Bell itself has produced.

Because the early compilers only provided sign extension, and the unsigned data type was only gradually hacked in, there has been a great deal of inconsistency in the implementation of the unsigned data type and the cast. In order to make C a more portable language something must be done.

What Can Be Done

The principle problem that must be dealt with is that many compilers don't support, or have limited support for, unsigned data types. In general it is easy to get the compiler to sign extend because that is the historical antecedant: all the compilers I have looked at, sign extend by default when converting a char to an int. I do know of a number of exceptions. But because of the inconsistency in the various compilers, if you want to zero extend, you had best do it yourself. This can be done by bitwise ANDing the char with 0377 to convert it to a short, or ANDing an int with 017777 to convert it to a long.

The big advantage to doing this rather than trying to rely on consistent compilers in the future is that you can be confident that it will always work, even if you are using one of the very old compilers that doesn't have unsigned at all. In fact, much of the UNIX* system and its utilities are written this way. It may not generate quite as good code in some compilers, but at least you are portable.

The problem remains with the perverse compilers that do not do sign extension. I have only painful things to say to the user of such a compiler who needs sign extension.

1) I think the compiler you are using should not be called a C compiler. Thus Honeywell-IBM users are in trouble.

2) You really should fix the compiler, or try to get it fixed.

3) You are going to have to write code to do the sign extension explicitly:

```
#define CTOS(x)  ((x&0200)?(x|0177400):(x&0377))
#define STOL(x)  ((x&0100000)?(x|037777600000L):(x&0177777L))
```

*UNIX is a Trademark of Bell Laboratories.

Just for completeness here is the code to zero extend, as I described it above:

```
#define ZCTOS(x) (x&0377)
#define ZSTOL(x) (x&0177777L)
```

What I would like to see done

The solution is twofold:

1) Fix all the compilers for all the machines so they are completely consistent.

2) Fix the white book so it clearly defines exactly at what points sign extension and zero extension are done, and change the definition so that there is no machine dependency.

I am making all the Microsoft C compilers obey the Bell standard. They mostly do already, much more so than either of the V7 compilers. I have come up with a proposed new definition that I think is clearer and more easily applied than what is currently in the white book. It would involve altering Appendix A, _The C Reference Manual_.

Appendix A, Section 6.1 should be rewritten to say that chars are signed, unsigned chars are unsigned.

The relevant sections of Appendix A, 6.5 and 6.6 should be replaced by the following:

0) _Definitions_
An _operation_ may be either arithmetic or logical (bit-wise)
An _integer_ may be either signed or unsigned.

1) The types _char_, _int_, _short_, and _long_ are signed.
The types _unsigned char_, _unsigned int_, _unsigned short_, _unsigned long_, and _unsigned_ are unsigned. A pointer to anything is unsigned.

2) No integer operation is done in a type shorter than the length of _int_. No integer operation is done in a type shorter than the longest operand.

3) When combining an unsigned and a signed integer in an operation, the result is unsigned.

4) When converting a shorter type to a longer type, if the source type is unsigned, the source is zero padded to the length of the longer type. If the source type is signed, it is sign extened.

5) When converting a longer type to a shorter type, high order bits are ignored, and the low order bits are considered to be of the shorter type.

6) It is not necessary that all the conversions actually take place. It is only necessary that results be as if all conversions had taken place.

Note that I don't address floating point here. Even though the white book says that floating point tends to be fairly machine dependent, (and believe me, it is) for some reason it hasn't been much of a problem. I suspect this is because the problems are mostly with loss of precision, rather than blatantly wrong results. I also suspect most people that really care about floating point use FORTRAN.

The other issue I don't address is machines which encourage character sets that take up 8 bits, and have useful characters with the high order bit set. CDC and IBM have such machines. It seems to me the only option open is to preserve the rules above, and to drop the restriction that chars have to be in the positive range of the character. Then we would have to insist that users playing the usual char-'0' tricks declare their characters unsigned if they want to be portable. These tricks dont work in EBCDIC anyways. I know very little about the the use of C on such machines. Perhaps it is unimportant.

DEAR ABBY DEAR ABBY DEAR ABBY DEAR ABBY DEAR ABBY DEAR ABBY DEAR ABBY DEAR ABBY

My mailbag this month is full of letters on the "Californian programmer" controversy. Here are samples from both sides of the debate:

> From andrewt Thu Nov 18 15:19:29 1982
> To: abby
> Subject: Berkley Software
>
> The members of the Basser Department of Computer Science seem to have a very negative attitude towards Unix software produced by the Unversity of California at Berkley. Whereas the software dis- tributed from Bell Laboratories seems to be regarded as the best thing since sliced bread. Why do the lecturers and professional staff of Basser propagate such ideas? It is producing a very unhealthy bias in the students at Basser.
>
> From karlos Fri Nov 19 16:54:13 1982
> To: abby
> Subject: 4.3bsd
>
> May I expressed in the harshest terms imaginable my displeasure at the callous individuals at Basser who have been criticising those fine lads at UCB.
>
> They seem to forget that the only machines that we will use in the future are VAX-11s, or machines whose C compilers we can coerce into vax lookalikes.
>
> They forget that ints are 32 bits, as are pointers, and that asm's are an important part of modern computing. After all what does the compiler produce, Algol 68?
>
> Who really cares if the 4.2 file system is quadratic when greater than 80% full. We all have enough money to have disk drives to burn.
>
> Fair go, fellas!
>
> Karlos Movcee III.

I think there is a natural tendency for Sydneysiders to side with the "original and best" at Bell, as a sort of East coast allegiance across continents. Also, more people there speak an intelligible dialect of English.

Indeed, those people who are shocked and offended by the expressions and statements produced in Californian speech are often those shocked by the expressions, statements and declarations produced in Californian code. Having watched the decline and fall of the English language from our cinema seats, why should we expect Californians to be able to cope with the exacting tyrannies of BNF syntax and unerring automata? As Djikstra points out, programming is the art of writing essays in crystal clear prose and making them executable.

Many theories have been put forward to explain the bizzare creatures which inhabit the State of California. One school observes that the people who moved away from the original colonies on the East Coast were the no-hopers, the criminals, the lunatic fringe and the religious nuts. Some of them stopped before the Rocky Mountains, but the real weirdos were pushed further west until they met the constraint of the Pacific Ocean. Thus the genetic debris of the human race found its stockpile in California. Or so the story goes.

Another factor was the 19th Century gold rushes, which attracted desperate and unsucessful men from all over the world, weeded out the weaker ones, and gave many of them money and power on a worse than random basis.

Perhaps we are seeing this phenomenon repeated with the high-tech gold rush in Silicon Valley. The difference is that instead of rewarding the healthy and hardy, those who succeed in the mind-bending business of software creation are often monomanic psychopaths.

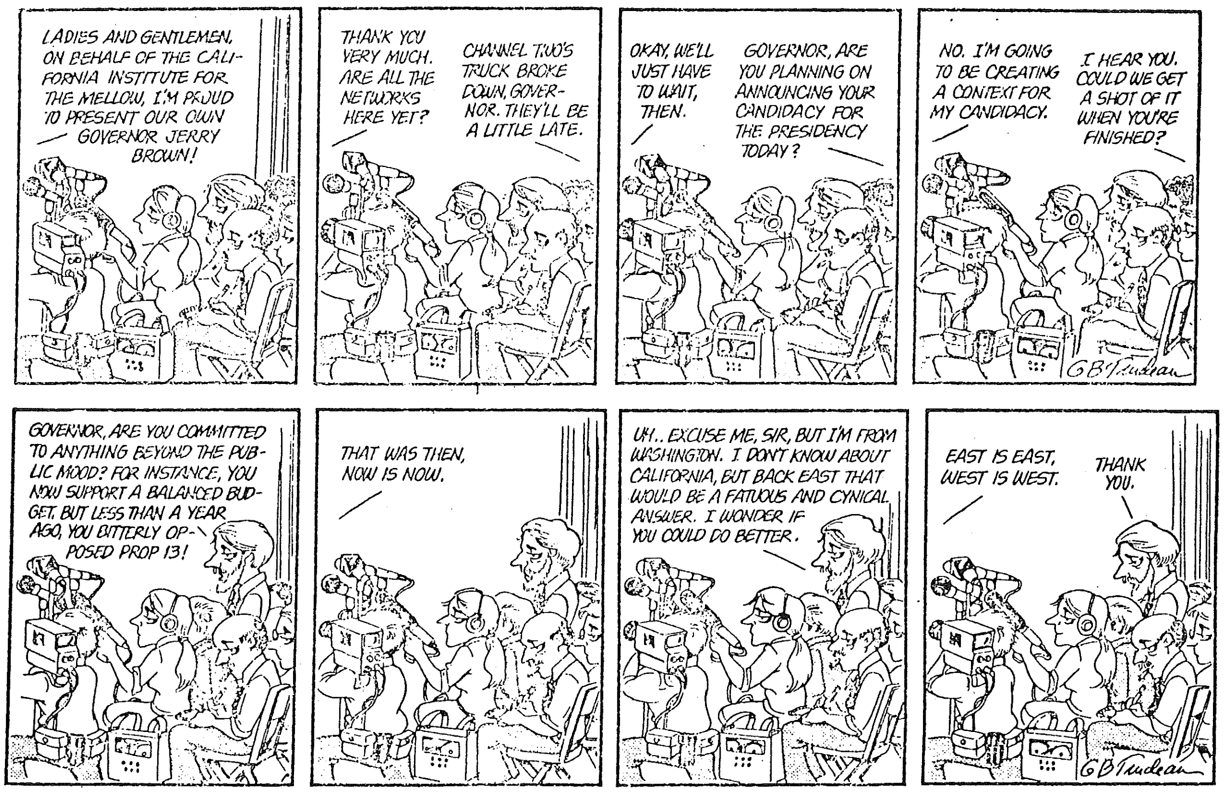
Some people believe that the root of all the madness is the same poison which undermined Ancient Rome -- lead. Or perhaps it's the brain fluids spinning around their heads from too many turns on the clover leaf highways. Witness the premier product of Pacific Pallisades -- Ronald Reagan.

Some medical reports indicate that Californians have a high sperm count in their cerebro-spinal fluid, a condition named "the FITH syndrome" (for F***ed In The Head).

The situation can be expected to worsen as the better people leave California for cleaner, safer hi-tech areas such as North Carolina, Boston, Oregon, Colorado, and even Texas. Many Portland cars publicise a worry of their owners with the bumper sticker "Don't Californicate Oregon".

It seems that little can be done about the problem except to hope that the government maintains its extremely liberal gun laws. As to the cause of this strange state of affairs and its de facto capital of San Franciso, the psycho-linguists and anthropologists can only speculate. Maybe they've just been out in the sun too long.

Abby


LADIES AND GENTLEMEN, ON BEHALF OF THE CALIFORNIA INSTITUTE FOR THE MELLOW, I'M PROUD TO PRESENT OUR OWN GOVERNOR JERRY BROWN!
THANK YOU VERY MUCH. ARE ALL THE NETWORKS HERE YET?
CHANNEL TWO'S TRUCK BROKE DOWN, GOVERNOR. THEY'LL BE A LITTLE LATE.
OKAY, WE'LL JUST HAVE TO WAIT, THEN.
GOVERNOR, ARE YOU PLANNING ON ANNOUNCING YOUR CANDIDACY FOR THE PRESIDENCY TODAY?
NO. I'M GOING TO BE CREATING A CONTEXT FOR MY CANDIDACY.
I HEAR YOU. COULD WE GET A SHOT OF IT WHEN YOU'RE FINISHED?
GB Trudeau
GOVERNOR, ARE YOU COMMITTED TO ANYTHING BEYOND THE PUBLIC MOOD? FOR INSTANCE, YOU NOW SUPPORT A BALANCED BUDGET, BUT LESS THAN A YEAR AGO, YOU BITTERLY OPPOSED PROP 13!
THAT WAS THEN, NOW IS NOW.
UH.. EXCUSE ME, SIR, BUT I'M FROM WASHINGTON. I DON'T KNOW ABOUT CALIFORNIA, BUT BACK EAST THAT WOULD BE A FATUOUS AND CYNICAL ANSWER. I WONDER IF YOU COULD DO BETTER.
EAST IS EAST, WEST IS WEST.
THANK YOU.
GB Trudeau

Date: 3 Dec 1982 1657 (Friday)
From: dave:csu40
To: timl:basservax auugn:basservax kev:elec70a lindsay:agsm
Subject: C compiler

Just been looking at the new VMS C compiler (!). It has an interesting feature in it which I think should receive consideration:

You can include files with £module <name> whereby a default library is searched for the module named. This has the advantage of saving disk space, whereby many small (< 1 block) files are coalesced into a library

I propose something for UNIX along these lines:

```
£module <fred.h>                  looks for fred.h in /usr/include/lib.a

£module <anotherlib(name)>        looks for name in /usr/include/anotherlib.a

£module <sys/lib(name)>           look in /usr/include/sys/lib.a

£module <directory/(name)>        default library, explicit path
```

Note that the directory prefix can still be given with -I....

If there's enough interest, I'll write it up more fully for AUUGN.

Dave Horsfall

Date: 2 Dec 1982 1508 (Thursday)
From: root:csu40
To: netgurus:basservax
Subject: name change

Because silly DECNET/VMS allows only 6 characters in a node name, I have had to change "csuvaxl" to "csuvxl". It's pathetic, I agree.

I am working on a simple DECNET/SUN gateway, comunicating via MAIL.

Dave Horsfall

Date: 6 Dec 1982 1319 (Monday)
From: chris
To: auugn
Subject: C compiler

Why does it have to be called '£module' ?
Surely it suffices to expand the syntax of '£include' to allow ar libraries of include files ala your suggestion (or make). EG

```
        £include        <lib(entry.h)>
```

Personnally, I don't think it is particularly useful, and is bound to cause annoyance when programs using it get distributed.

AUUGN is produced by volunteers from the Australian Unix Users Group. Material on Unix is solicited from readers. Please send your contributions to the editors at the address given below.

The subscription fee for AUUGN is $24 for six issues over one year. Overseas subcription is $30 Australian dollars. Please do not send purchase orders, only money! The volunteers running AUUGN don't like paper warfare! Your subscription fee and contributions should be mailed to:

AUUGN
c/o Bob Kummerfeld
Basser Department of Computer Science
University of Sydney
AUSTRALIA 2006