Historical Note: This Document was recovered from an original printed in August, 1977. – Tom Lyon

Inter-UNIX Portability

Thomas L. Lyon

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Having the UNIX operating system on many different computers presents a challenge in writing C programs which are portable between UNIX systems, regardless of machine. Portability guidelines are presented which were formulated by porting the most common UNIX programs to the Interdata 8/32. Statistics are presented which outline benefits and disadvantages of portability.

Introduction

Although much work remains to be done, it can be said that the project to bring up the UNIX system on the Interdata 8/32 has been successful. During the course of this project, much has been learned about what is necessary for a C program to be machine independent. This author's working definition of machine independence is that there should exist a single copy of the source of a program which can be compiled by different compilers, and, with possibly differing header files, be run on any machine running a UNIX system. Note that this last condition is not at all trivial; it has proved to be harder to move a program from a UNIX system to another system on the same machine than to move from UNIX on one machine to UNIX on another machine. This paper will give details of problems encountered in porting programs, guidelines for writing or rewriting programs to be portable, and some statistics about the advantages and disadvantages of portable programming.

Problem Areas

1. Word Length

Perhaps the most obvious difference between machines is in the number of bits in a word. In practice, programs which depend on word length have been easy to recognize and easy to fix. Word length problems are most often manifested in programs which play with bits, such as those using bit maps or those using bit-wise operators, as in

```
x & 0177770
```

which will clear the low order 3 bits on 16 bit machines, but will clear others as well on machines with larger words. A better approach would be

```
x & ~07
```

Many applications could get rid of this clumsiness altogether by using the bit field feature of C, although even this must be used carefully (see section 7).

2. Signed vs. Unsigned Characters

The PDP-11 has the peculiarity that the high order bit of characters is treated as a sign bit, whereas most other machines, including the Interdata, allow only positive values for characters. The most common portability problem involving this difference occurs with a program segment such as

```
char c;
if((c=getchar()) == -1) ...
```

in which the comparison for -1 will fail on any machine but the PDP-11 since characters must be positive. The variable c should have been declared as an int, because *getchar* is actually an integer valued function; it can return the value -1 as well as any character value.

3. Unsigned vs. Signed Pointer Comparisons

On the PDP-11, pointers are treated as unsigned quantities when they are compared, but on the Interdata, they are treated as signed. This would not cause portability problems if strict type checking were used, but can (and has) lead to difficulties otherwise. For example, one can assign -1 to a pointer on the PDP- I I and then use it as the maximum possible pointer value, but on the Interdata this pointer would be less than all valid pointers. The solution to this problem is to use only valid pointers instead of creating invalid ones.

4. Pointer Wrap-around

It is possible in C that high order bits may be lost during a pointer computation. Although this problem has never occurred in moving from the PDP- 11 to the Interdata, it has happened in moving from the PDP- I I to another machine and. operating system. A typical program segment vulnerable to this problem is

```
struct a x[100], *p;
 for (p = x; p < &x[100]; p ++) ...
```

Since the object x[100] does not actually exist, wrap-around could occur in computing its address so that &x[100] was actually less than &x[0]. This particular problem would only occur if the array x were at the end of memory; it is much more important to worry about programs which use subscripts which are negative or clearly out of range.

5. Floating Point

Since almost every machine has its own style of floating point architecture, it is not surprising that this could be a problem area in portability. Fortunately, however, about the only problem that arises is that of tolerances: the results of floating computations may be equal on one machine but only almost equal on another machine. In practice, this would seldom arise because defensive programming seems to be the rule in floating comparisons.

6. Shifting

Machine anomalies crop up in right shifting. On the PDP-11, int quantities are shifted arithmetically, but on the Interdata they are shifted logically. However, unsigned quantities are guaranteed by the C language definition to be shifted logically, so if there is any possibility of trouble, int values should be cast to unsigned before shifting.

Another problem with shifting is that the maximum shift count varies from machine to machine. Thus, 1L<<32 has the value 0 on the PDP-11 since the I bit was shifted into oblivion, but on the Interdata, the result is 1 because only the low order 5 bits of the shift value are used, and so no shifting took place.

7. Byte Order

The order of bytes within a word is the problem which has created the most trouble in moving from the PDP-11 to the Interdata 8/32. On the PDP-11, the byte with the lower address is the low-order byte of the word, and the byte with the higher address is the highorder part of the word. On the Interdata, the reverse is true: the lowest address byte is the highest order byte of a word and the highest address byte is the lowest order byte of a word. This problem most often shows up in

communication between the two machines, but can also appear in single machine situations.
Consider the following familiar function.

```
putchar(c)
{
        write(1, &c, 1);
}
```

Since the variable c is not declared, it is assumed to be an int. Thus, the second argument passed to write is the address of an integer, which on the PDP-11 is also the address of the character contained in that integer. But on the Interdata, the address passed is that of the high order byte of the integer rather than that of the low order byte which is where the character is. Thus, only null characters would ever be written. This particular example would work if c were declared as a **char;** in general if **int and char** data are mixed, the **union** construct should be used.

A related problem is that the ordering of bit fields within a word is the same as the ordering of bytes. So, on the PDP-11, the first field is in the lowest order bits of the word; on the Interdata, the first field is in the highest order bits. This problem should affect only those programs which depend on the relative positions of bit fields (unfortunately, most do).

The byte-ordering differences create very serious problems in machine to machine communication. Because of the nature of I/O, all communication between the PDP-11 and the Interdata 8/32 is in a byte-by-byte fashion. Thus, if one transmits character data to the other machine, that data is reconstructed in the other machine in the proper order. However, if one sends integer data, it is first broken down into byte-by-byte order and when the data is reconstructed on the other machine the bytes of the integer are in the wrong order. There does not exist any easy solution to this problem. If one must transmit integer data from one machine to another, the exact format of that data must be known so that the appropriate bytes may be swapped either before or after transmission.

Portability Guidelines

During the course of the UNIX portability project, it has become apparent that the best criterion for ease of portability is that programs should be written in good style. Now, "good style" is admittedly hard to define, and this definition may have been influenced by experiences in portability; but nonetheless, it has been this author's overwhelming experience that a stylistically pleasing program is an easily portable program. With regard to this, the following sections will discuss techniques which will vary from being necessary for machine independence to being suggestions for better style.

1. Avoid Efficiency Tricks

Many of the problems encountered in porting a program are due to efficiency tricks, i.e., places where the programmer has taken advantage of his knowledge of the machine's architecture to give a slight efficiency boost to his program. Such constructs are almost always nonportable, and should be replaced by constructs which are guaranteed by the C language definition to work on any machine. Indeed, it could be argued that a C programmer should never know what machine his program will run on, so that he writes code strictly by the C manual instead of by the machine manual.

2. Use Header Files

One of the most important steps in writing a portable program is to use the C preprocessor features of include files and definitions. Whenever definitions of data types, formats, or values are shared by multiple programs, include files should be used to centralize these definitions and to assure that there is only one copy to change if change is ever needed. The directory /usr/include contains include files intended for public use. A line of the form "#include <xxx.h>" will include the file /usr/include/xxx.h. Typical of these files are <dir.h>, a definition of the structure of a directory, and <signal.h>, a definition of system signal numbers. Another important use of header files is to isolate environment dependent data, such as default file names or options. In

fact, anything that could vary between systems, even if they are on the same type of computer, should be placed in a header file where it may be easily located and modified.

3. Isolate Constants

With few exceptions, whenever a constant appears in the body of a program one can be justifiably worried about that program's portability or style. Innocent looking constants like 0, 1, and 2 are sometimes exceptions, but if one found a 79 in a program one would have to stop and ask what that 79 really means, how it was derived, and why it isn't expressed in terms which make its meaning obvious. For instance, programs often maintain state variables which take on a number of values depending on the state of the program. These values are best expressed by using the #define feature of the preprocessor to associate names with the values, instead of using numeric constants in the code which have no immediate meaning.

Often a constant is used to represent the size of some object; such usage is not at all portable and should be changed to use the sizeof operator of C. If a program uses constants as array subscripts, then it is likely that each element of that array has some different type of meaning. Such array uses could best be replaced by structures, which not only have a different meaning for each element, but also have a different name for each element to make that meaning apparent. The most common example of this misuse of arrays is in the stty and gtty system calls, which, on the PDP-11, are commonly invoked with a three word array as an argument, instead of the structure defined in /usr/include/sgtty. h. This usage is also non-portable since the structure is only two words on the Interdata. Another type of constant to watch out for is that of constant file names. Such file names are almost always environment dependent and should be isolated in a header file or #define statement.

4. Use New C Features

Perhaps the greatest help in writing portable C programs is given by some of the newer features of the language. After studying the needs of portability, these features were chosen to allow ease of expression and portability in what are normally messy areas. The **typedef** feature allows the programmer to isolate the definition of machine dependent data types, while the **sizeof(type)** construct allows finding the size of any of these defined data types. The **union declaration** allows the overlaying of data, which has historically been a very trying problem in portability.

5. Use Canned Routines

A wise rule to follow in programming is: don't re-invent the wheel. One should use publicly available canned routines as much as possible-, this reduces the possibility of duplicated source code, leads to increased modularization, makes programming easier, and makes maintenance of any particular function easier.

It is not at all uncommon to see programs which contain functions which do exactly the same thing as publicly available functions-, *atoi* and *strcmp* are prime examples. The programmer should take it upon himself to be acquainted with recent additions to the C library. The most important package of routines as far as portability is concerned is the standard I/O library, developed as a replacement and enhancement to the portable I/O library. The *getc* and *putc* macros provided by this package should be used to replace the old *getc* and *putc* routines of the PDP-11**,** since these routines do not exist for the Interdata 8/32. Routines which have proven particularly useful in revision of programs to remove duplicated function include *sprintf, fgets* and *fputs*.

6. Use Lint

The lint command, a C program checker, has proved to be an invaluable aid to porting programs. Used with the strict type checking option (-s flag), lint provides very useful information about possibly non-portable constructs. If one were to use and respect lint at every stage of a program's development, that program would be very easily portable to another UNIX system; there would be a good chance that no changes would be needed to port it. When converting an old program into a portable program output from lint is very useful in identifying trouble spots. On the other hand, the output from lint is often extremely voluminous; rethinking the data structures often does wonders to shut up complaints about strict type checking.

Statistics

This section will give some estimate of the cost of writing portable C programs. Of about 220 programs in the /usr/src/cmd directory, over 80 have been converted into portable programs which will run on either the PDP-11 or the Interdata 8/32. This represents about 19000 of 45000 lines of C. In general, it has been observed that when a program was converted into a portable program, the size of the source code decreased and the size of the object code increased.

The average decrease in source size was 9%, while the total amount of source also decreased by 9%. This decrease was caused by the use of more canned routines and header files, as suggested in the previous section. The source size is defined as the number of lines in the program.

The average increase in object size was 56%; total object size increased by 20%. Object size is defined as the total given by the size program. This isn't as bad as it seems. The huge average increase is due to the use of the standard 1/0 library in otherwise small programs. For instance, the *echo* program grew by a whopping 880%, but this is actually only 2078 bytes. The total increase of 20% is a true cost of the style and portability improvements; this cost, however, is completely cancelled by the gain in case of maintenance. Having only one copy of a program's source prevents major administrative headaches, and the improvements in style make understanding and modifying programs much easier.

The appendix gives a program-by-program breakdown of sizes before and after conversion to portable code.

Appendix

This appendix gives new and old, source and object sizes for a number of individual programs. The programs shown are those for which it was easy to collect statistics; these programs are not all that were converted, but are representative of those that were.

Size of Source Code (lines)

Command	Old Size	New Size	% Diff
ac	245	243	-0.82
ar	617	620	0.49
chown	95	54	-43.16
clri	77	79	2.60
cmp	125	120	-4.00
Comm	180	167	-7.22
cron	242	241	-0.41
date	201	152	-24.38
dcheck	249	220	-11.65
dd	536	533	-0.56
df	104	97	-6.73
diff	664	636	-4.22
echo	32	21	-34.38
getty	248	220	-11.29
ind	23	25	8.70
init	299	295	-1.34
kill	38	40	5.26
mail	429	338	-21.21
mesg	57	52	-8.77
mkfs	589	583	-1.02
mount	64	65	1.56
mv	155	184	18.71
ncheck	358	328	-8.38
newgrp	159	55	-65.41
nice	40	43	7.50
od	628	239	-61.94
passwd	197	156	-20.81
pr	442	414	-6.33
rev	44	44	0.00
rm	87	118	35.63
size	43	46	6.98
sort	813	823	1.23
split	88	81	-7.95
strip	117	ill	-5.13
stty	322	284	-11.80
su	123	38	-69.11
tabs	265	246	-7.17
tail	162	161	-0.62
tee	63	64	1.59
test	141	136	-3.55
time	129	93	-27.91
tr	144	144	0.00
uniq	173	141	-18.50
wall	70	66	-5.71
who	88	51	-42.05
write	176	181	2.84

Size of Object Code (total bytes)

Command	Old Size	New Size	% Diff
ac	10154	10258	1.02
ar	8492	10136	19.36
chmod	2428	3968	63.43
chown	3634	6028	65.88
clri	2854	2854	0.00
Cmp	2950	4704	59.46
Comm	5362	5274	-1.64
cron	3620	6122	69.12
date	2640	2852	8.03
dcheck	11888	11972	0.71
dd	5108	7294	42.80
df	2156	4362	102.32
diff	9320	9616	3.18
echo	236	2314	880.51
getty	1170	1198	2.39
ind	1612	3650	126.43
init	3620	3612	-0.22
kill	2350	2350	0.00
mail	10450	10746	2.83
mesg	1080	3884	259.63
mkfs	11024	11000	-0.22
mount	3886	5490	41.28
mv	3336	5602	67.93
ncheck	41306	41370	0.15
newgrp	8542	10594	24.02
nice	2650	4714	77.89
od	5278	6104	15.65
passwd	6476	10194	57.41
pr	12204	15632	28.09
rev	4100	4100	0.00
rm	4148	4452	7.33
size	4100	4128	0.68
sort	10222	10256	0.33
split	2252	4344	92.90
strip	2968	2978	0.34
Stty	2498	4794	91.91
su	6694	8504	27.04
tabs	1758	3346	90.33
tail	5730	5728	-0.03
tee	1960	1960	0.00
test	1420	4538	219.58
tr	4240	4282	0.99
uniq	5012	6622	32.12
wall	6108	8478	38.80
who	5962	5962	0.00
write	5966	5984	0.30