Hardware dependencies

The days are gone when moving a package from one hardware platform to another meant rewriting the package, but there are still a number of points that could cause you problems. In this chapter, we'll look at the most common causes.

Data types

All computers have at least two basic data types, characters and integers. While European languages can get by with a character width of 8 bits, integers must be at least 16 bits wide to be of any use, and most UNIX systems use 32 bit integers, as much storage as four characters. Problems can obviously arise if you port a package to a system whose `int` size is less than the author of the package expected.

Integer sizes

Data sizes aren't the problem they used to be—times were when a machine word could be 8, 12, 16, 18, 24, 30, 32, 36, 48, 60, 64 or 72 bits long, and so were the primary integer data objects. Nowadays you can expect nearly every machine to have an *int* of 16, 32 or 64 bits, and the vast majority of these have a 32 bit *int*. Still, one of the biggest problems in ANSI C is the lack of an exact definition of data sizes. *int* is the most used simple data type, but depending on implementation it can vary between 16 and 64 bits long. *short* and *long* can be the same size as *int*, or they can be shorter or longer, respectively. There are advantages to this approach: the C compiler will normally choose an `int` which results in the fastest processing time for the processor on which the program will run. This is not always the smallest data size: most 32-bit machines handle 32 bit arithmetic operations faster than 16 bit operations. Problems don't arise until the choice of `int` is too small to hold the data that the program tries to store in it. If this situation arises, you have a number of options:

- You can go through the sources with an editor and replace all occurrences of the word `int` with `long` (and possibly `short` with `int`).*

* If you do this, be sure to check that you don't replace `short int` with `int int`!

- You can simplify this matter a little by inserting the following definition in a common header file:

  ```
  #define int long
  ```

  This has the disadvantage that you can't define `short` as `int`, because preprocessor macros are recursive, and you will end up with both `int` and `short` defined as `long`.

- Some compilers, particularly those with 16-bit native `int`s, offer compiler flags to generate longer standard `int`s.

All these "solutions" have the problem that they do not affect library functions. If your system library expects 16-bit integers, and you write

```
int x = 123456;
printf ("x is %d\n", x);
```

the library routine `printf` still assumes that the parameter `x` is 16 bits long, and prints out the value as a signed 16-bit value (-7616), not what you want. To get it to work, you need to either specify an alternate library, or change the format specification to `printf`:

```
int x = 123456;
printf ("x is %l\n", x);
```

There are a few other things to note about the size of an *int*:

- Portable software doesn't usually rely on the size of an *int*. The software from the Free Software Foundation is an exception: one of the explicit design goals is a 32-bit target machine.
- The only 64-bit machine that is currently of any significance is the DEC Alpha. You don't need to expect too many problems there.
- 16 bit machines—including the 8086 architecture, which is still in use under MS-DOS—are a different matter, and you may experience significant pain porting, say, a GNU program to MS-DOS. If you really want to do this, you should look at the way *gcc* has been adapted to MS-DOS: it continues to run in 32-bit protected mode and has a library wrapper* to allow it to run under MS-DOS.

Floating point types

Floating point data types have the same problems that integer types do: they can be of different lengths, and they can be big-endian or little-endian. I don't know of any system where ints are big-endian and floats are little-endian, or vice-versa.

Apart from these problems, floats have a number of different structures, which are as good as completely incompatible. Fortunately, you don't normally need to look under the covers: as long as a float handles roughly the same range of values as the system for which the program was written, you shouldn't have any problems. If you do need to look more carefully, for example if the programmer was making assumptions, say, about the position of the sign bit of

* A *library wrapper* is a library that insulates the program (in this case, a UNIX-like application) from the harsh realities of the outside world (in this case, MS-DOS).

the mantissa, then you should prepare for some serious re-writing.

Pointer size

For years, people assumed that pointers and *int*s were the same size. The lax syntax of early C compilers didn't even raise an eyebrow when people assigned *int*s to pointers or vice-versa. Nowadays, a number of machines have pointers that are not the same size as *int*s. If you are using such a machine, you should pay particular attention to compiler warnings that *int*s are assigned to pointers without a cast. For example, if you have 16-bit `int`s and 32-bit pointers, sloppy pointer arithmetic can result in the loss of the high-order bits of the address, with obvious consequences.

Address space

All modern UNIX variants offer *virtual memory*, though the exact terminology varies. If you read the documentation for System V.4, you will discover that it offers virtual memory, whereas System V.3 only offered *demand paging*. This is more marketspeak than technology: System V.2, System V.3, and System V.4 each have very different memory management, but we can define *virtual memory* to mean any kind of addressing scheme where a process address space can be larger than real memory (the hardware memory installed in the system). With this definition, all versions of System V and all the other versions of UNIX you are likely to come across have virtual memory.

Virtual memory makes you a lot less dependent on the actual size of physical memory. The software from the Free Software Foundation makes liberal use of the fact: programs from the GNU project make no attempt to economize on memory usage. Linking the *gcc* C++ compiler *cc1plus* with GNU *ld* uses about 23 MB of virtual address space on System V.3 on an Intel architecture. This works with just about any memory configuration, as long as

- Your processes are allowed as much address space as they need (if you run into trouble, you should reconfigure your kernel for at least 32 MB maximum process address space, more if the system allows it).
- You have enough swap space.
- You can wait for the virtual memory manager to do its thing.

From a configuration viewpoint, we have different worries:

- Is the address space large enough for the program to run?
- How long are pointers? A 16 bit pointer can address only 64 kilobytes, a 32 bit pointer can address 4 GB.
- How do we address memory? Machines with 16 bit pointers need some kind of additional hardware support to access more than 64 kilobytes. 32 bit pointers are adequate for a "flat" addressing scheme, where the address contained in the pointer can address the entire virtual address space.

Modern UNIX systems run on hardware with 32 bit pointers, even if some machines have *int*s with only 16 bits, so you don't need to worry much about these problems. Operating systems such MS-DOS, which runs on machines with 16 bit pointers, have significant problems as a result, and porting 32 bit software to them can be an adventure. We'll touch on these problems in Chapter 20, *Compilers*, page 346.

Character order

The biggest headache you are likely to encounter in the field of hardware dependencies is the differing relationship between *int* and character strings from one architecture to the next. Nowadays, all machines have integers large enough to hold more than one character. In the old days, characters in memory weren't directly addressable, and various tricks were employed to access individual characters. The concept of byte addressing, introduced with the IBM System/360, solved that problem, but introduced another: two different ways of looking at bytes within a word arose. One camp decided to number the bytes in a register or a machine word from left to right, the other from right to left. For hardware reasons, text was always stored from low byte address to high byte address.

A couple of examples will make this more intelligible. As we saw above, text is always stored low byte to high byte, so in any architecture, the text "UNIX" would be stored as

0	1	2	3
U	N	I	X

Some architectures, such Sparc and Motorola 68000, number the bytes in a binary data word from left to right. This arrangement is called *big-endian*. On a big-endian machine, the bytes are numbered from left to right, so the number 0x12345678 would be stored like

0	1	2	3
12	34	56	78

Others, notably older Digital Equipment machines and all Intel machines, number the bytes the other way round: byte 0 in a binary data word is on the right, byte 3 is on the left. This arrangement is called *little-endian*.* The same example on a little-endian machine would look like:

3	2	1	0
12	34	56	78

This may look just the same as before, but the byte numbers are now numbered from right to left, so the text now reads:

* The names *big-endian* and *little-endian* are derived from Jonathan Swift's "Gulliver's Travels", where they were a satirical reference to the conflicts between the Catholics and the Church of England in the 18th Century.

3	2	1	0
X	I	N	U

As a result, this phenomenon is sometimes called the *NUXI*[*] syndrome. This is only one way to look at it, of course: from a memory point of view, where the bytes are numbered left to right, it looks like

0	1	2	3
78	56	34	12

and

0	1	2	3
U	N	I	X

It's rather confusing to look at the number 0x12345678 as 78563412, so the NUXI (or XINU) view predominates. It's easier to grasp the concepts if you remember that this is all a matter of the mapping between bytes and words, and that text is *always* stored correctly from low byte to high byte.

An alternative term for *big-endian* and *little-endian* is the term *byte sex*. To make matters even more confusing, machines based on the MIPS chips are veritable hermaphrodites—all have configurable byte sex, and the newer machines can even run different processes with different byte sex.

The problem of byte sex may seem like a storm in a teacup, but it crops up in the most unlikely situation. Consider the following code, originally written on a VAX, a little-endian machine:

```
int c = 0;

read (fd, &c, 1);
if (c == 'q')
  exit (0);
```

On a little-endian machine, the single character is input to the low-order byte of the word, so the comparison is correct, and entering the character `q` causes the program to stop. On a 32-bit big-endian machine, entering the character `q` sets `c` to the value 0x71000000, not the same value as the character `q`. Any good or even mediocre compiler will of course warn you if you hand the address of an `int` to `read`, but only if you remember to include the correct header files: it happens anyway.

* Why not *XINU*? Because the term arose when words were 16 bits long. The PDP-11, for example, stored `int`s (16 bit quantities) in a little-endian format, so pairs of bytes were swapped. The PDP-11 also had 32 bit `long` quantities that were stored with their component words in a big-endian format. This arrangement has been called `mixed-endian`, just to add to the general confusion.

This discussion has concentrated on how characters are ordered within words, but the same considerations also affect bit fi elds within a word. Most hardware platforms don't support bit fi elds directly: they're an idea in the mind of the compiler. Nonetheless, all architectures defi ne a bit order: some number from left to right, some from right to left. Well-written programs don't rely on the order of bit fi elds in *int*s, but occasionally you see register defi nitions as bit fi elds. For example, the 4.4BSD sources for the HP300 include the following defi nition:

```
struct ac_restatdb
  {
  short ac_eaddr;              /* element address */
  u_int ac_res1:2,
        ac_ie:1,               /* import enabled (IEE only) */
        ac_ee:1,               /* export enabled (IEE only) */
        ac_acc:1,              /* accessible from MTE */
        ac_exc:1,              /* element in abnormal state */
        ac_imp:1,              /* 1 == user inserted medium (IEE only) */
        ac_full:1;             /* element contains media */
  };
```

This defi nition defi nes individual bits in a hardware register. If the board in question fi ts in machines that number the bits differently, then the code will need to be modifi ed to suit.

Data alignment

Most architectures address memory at the byte level, but that doesn't mean that the underlying hardware treats all bytes the same. In the interests of effi ciency, the processor accesses memory several bytes at a time. A 32-bit machine, for example, normally accesses data 4 bytes at a time—this is one of the most frequent meanings of the term "32-bit machine". It's the combined responsibility of the hardware and the software to make it look as if every byte is accessed in the same way.

Confl icts can arise as soon as you access more than a byte at a time: if you access 2 bytes starting in the last byte of a machine word, you are effectively asking the machine to fetch a word from memory, throw away all of it except the last byte, then fetch another word, throw away all except the fi rst, and make a 16 bit value out of the two remaining bytes. This is obviously a lot more work than accessing 2 bytes at an even address. The hardware can hide a lot of this overhead, but in most architectures there is no way to avoid the two memory accesses if the address spans two bus words.

Hardware designers have followed various philosophies in addressing data alignment. Some machines, such as the Intel 486, allow unaligned access, but performance is reduced. Others, typically RISC machines, were designed to consider this to be a Bad Thing and don't even try: if you attempt to access unaligned data, the processor generates a trap. It's then up to the software to decide whether to signal a bus error or simulate the transfer—in either case it's undesirable.

Compilers know about alignment problems and "solve" them by moving data to the next address that matches the machine's data access restrictions, leaving empty space, so-called *padding* in between. Since the C language doesn't have any provision for specifying alignment information, you're usually stuck with the solution supplied by the compiler writer: the compiler automatically aligns data of speciﬁ c types to certain boundaries. This doesn't do much harm with scalars, but can be a real pain with *struct*s when you transfer them to disk. Consider the following program excerpt:

```
struct emmental
  {
  char flag;
  int count;
  short choice;
  int date;
  short weekday;
  double amount;
  }
emmental;
read_disk (struct emmental *rec)
  {
  if (read (disk, rec, sizeof (rec)) < sizeof (rec))
    report_bad_error (disk);
  }
```

On just about any system, `emmental` looks like a Swiss cheese: on an i386 architecture, *short*s need to be on a 2-byte boundary and *int*s and *double*s need to be on a 4-byte boundary. This information allows us to put in the offsets:

```
struct emmental
{
  char flag;                  /* offset 0 */
  /* 3 bytes empty space */
  int count;                  /* offset 4 */
  short choice;               /* offset 8 */
  /* 2 bytes empty space */
  int date;                   /* offset 12 */
  short weekday;              /* offset 16 */
  /* 2 bytes empty space */
  double amount;              /* offset 20 */
  }
emmental;
```

As if this weren't bad enough, on a Sparc *double*s must be on an 8-byte boundary, so on a Sparc we have 6 bytes of empty space after `weekday`, to bring the offset up to 24. As a result, `emmental` has 21 useful bytes of information and up to 13 of wasted space.

This is, of course, a contrived example, and good programmers would take care to lay the `struct` out better. But there are still valid reasons why you encounter this kind of alignment problem:

- If `flag`, `count` and `choice` are a key in a database record, they need to be stored in this sequence.
- A few years ago, even most good programmers didn't expect to have to align a *double* on an 8-byte boundary.

- A lot of the software you get looks as if it has never seen a good programmer.

Apart from the waste of space, alignment brings a host of other problems. If the first three fields really are a database key, somebody (probably the database manager) has to ensure that the gaps are set to a known value. If this database is shared between different machines, our `read_disk` routine is going to be in trouble. If you write the record on an i386, it is 28 bytes long. If you try to read it in on a Sparc, `read_disk` expects 32 bytes and fails. Even if you fix that, `amount` is in the wrong place.

A further problem in this example is that Sparcs are big-endian and i386s are little-endian: after reading the record, you don't just need to compact it, you also need to flip the bytes in the *short*s, *int*s and *double*s.

Good portable software has accounted for these problems, of course. On the other hand, if your program compiles just fine and then falls flat on its face when you try to run it, this is one of the first things to check.

Instruction alignment

The part of the processor that performs memory access usually doesn't distinguish between fetching instructions from memory and fetching data from memory: the only difference is what happens to the information after it has reached the CPU. As a result, instruction alignment is be subject to the same considerations as data alignment. Some CPUs require all instructions to be on a 32 bit boundary—this is typically the case for RISC CPUs, and it implies that all instructions should be the same length—and other CPUs allow instructions to start at any address, which is virtually a requirement for machines with variable length instructions.* As with data access, being *allowed* to make this kind of access doesn't make it a good idea. For example, the Intel 486 and Pentium processors execute instructions aligned on any address, but they run significantly faster if the target address of a jump instruction is aligned at the beginning of a processor word—the alignment of other instructions is not important. Many compilers take a flag to tell them to align instructions for the i486.

* Some machines with variable length instructions *do* have a requirement that an instruction fit in a single machine word. This was the case with the Control Data 6600 and successors, which had a 60 bit word and 15 or 30 bit instructions. If a 30 bit instruction would have started at the 45 bit position inside a word, it had to be moved to the next word, and the last 15 bits of the previous instruction word were filled with a *nop*, a "no-operation" instruction.