C no es un lenguaje inherentemente portable

10.2.3) Porting Software to ARM Linux

Many times the software you would like to run on the iPAQ is written in C. C is not an

inherently portable language. To write portable code in C generally requires some extra

thought.

There are some portability issues that may need some special attention when we run into

when porting applications to ARM Linux, especially from x86 Linux.

a) C Portability Issues

There are a number of areas in which the definition of a C program's behavior depend on

the architecture on which the program is run. It's behavior can depend on the peculiarities

of the OS, the compiler, the libraries, and the CPU.

b) Signed vs. Unsigned Characters

The C standard says that char may either be signed or unsigned by default. On

x86 Linux, char is signed by default. On ARM Linux, char is unsigned by default.

Comparing a char to a negative number will always return 0, because the char is

unsigned and therefore positive.

c) Pointer Alignment Issues

On many CPU architectures, the memory system requires that loads of values larger than

one byte must be properly aligned. Usually, this means that a 2-byte quantity must be

aligned on an even address boundary, a 4-byte quantity must be aliged on a multiple of 4

boundary and sometimes 8-byte quantities must be aligned to addresses that are a

multiple of 8. Depending on the CPU and the operating system, misaligned loads and

stores may cause a signal, may be handled in the OS, or may be silently rounded to the

appropriate boundary.

The x86 boundary imposes no such alignment restriction, so some programs written for

the x86 do not use the proper alignment for other architectures.

ARM Linux defaults to silently round the address to the appropriate alignment boundary.

d) Using Memory Overlays to Convert Types

This is very non-portable. The code has to be written so that alignment, size, and

endianness are all correctly handled across the supported architectures.

e) Endianness Issues

There are two basic memory layouts used by most computers, designated big

endian and little endian. On big endian machines, the most significant byte of an

object in memory is stored at the least signicant (closest to zero) address

(assuming pointers are unsigned). Conversely, on little endian machines. the

least significant byte is stored at the address closest to zero. Let's look at an

example:

int x = 0xaabbccdd;

unsigned char b = \*(unsigned char \*)&x;

On a big endian machine, b would receive the most significant byte of x, 0xaa.

On little endian machines, b would receive the least signficant byte of x: 0xdd.

The x86 architecture is little endian. Many ARM processors support either mode,

but usually are used in little endian mode.

Endian problems arise under two conditions:

• When sharing binary data between machines of different endianness.

• When casting pointers between types of different sizes

In the first case, the data appears in the correct location, but will be interpreted

differently by the different machines. If a little endian machine stored

0xaabbccdd into a location, a big endian machine would read it as 0xddccbbaa.

In the second case, on a little endian machine there is no problem: a char, short,

or int stored in an int sized variable each have the same address. On a big

endian machine, if you want to be able to store a short and then read it as an int

you have to increment the pointer so that the MSB lands in the right place.

REFERENCIA

Open Components for Embbeded Real Time Applications, OCERA User’s Guide, WP10 -D10.7 User Guide, Ocera\_UG.pdf, page 172 a 173/235.