Strong Engineering LLC

Handling Foreign Data in Embedded Systems

Duane Strong

Strong Engineering LLC

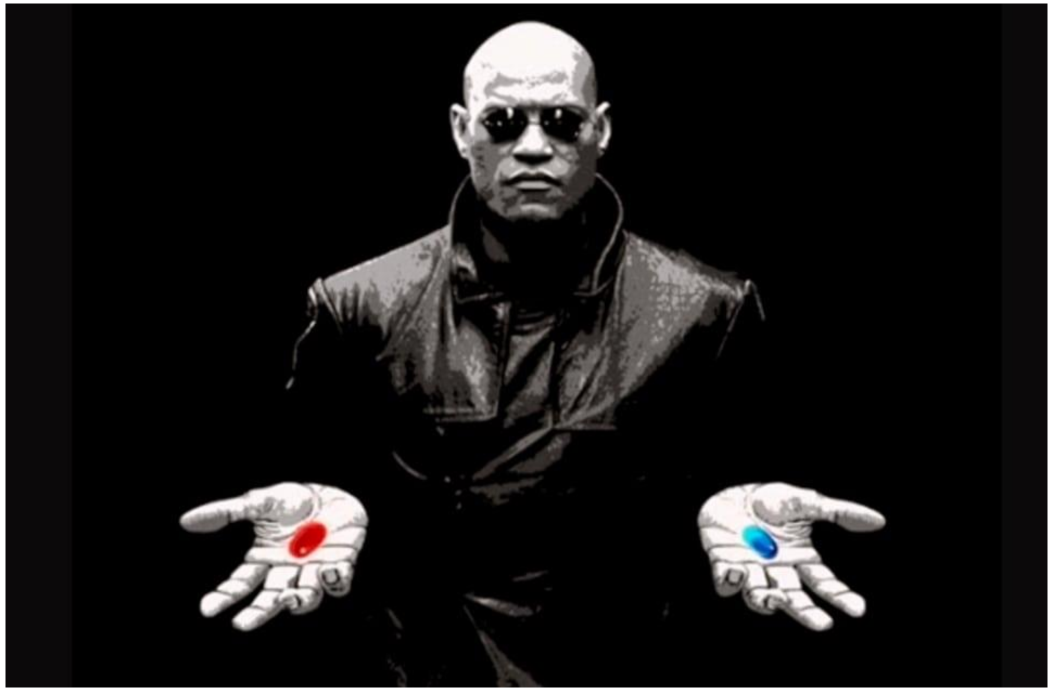
duanes@strongenging.com

Introduction

Every CPU architecture has its unique qualities. Some of these qualities involve how data is represented in memory. Some people may call these quirks. Most programmers are blissfully unaware of these quirks. Within a given system the CPU will create and consume its data in a consistent manner, effectively hiding these quirks. We could say that the CPU is “eating its own dogfood” and as a normal programmer we don’t have to do anything or stay awake at night worrying about it.

You took the red pill

But you, my friend, are not normal. You are an embedded programmer and have taken the red pill. You must embrace the uncomfortable truth.



How foreign data enters the system

We will call data that was not created by the CPU we are running on “foreign” data and it can enter into the normally closed system in a number of ways.

- **Transmitted** - Data can be received from the outside world via USB, SPI, i2c, serial, network, wireless or other means.
- **Mounted -** Data can be accessed via SD card, mounted file system, proprietary ROM modules etc.
- **Memory mapped -** Hardware can be mapped into the address space that is not native to the CPU architecture.

Data representation quirks

If all the data in the world were represented as simple arrays of bytes in memory, there wouldn’t be a problem.

All bytes are self-consistent in that they hold eight bits, arranged with the most significant bit on the left, and the least significant bit on the right, modelling how we represent place values in base 10.

I’m sure some wise guy tried to reverse that back in the day and were rightfully ostracized from society.

Even signed data has been universally agreed upon to be represented in twos-complement.

Data representation quirks

Unfortunately, people demand data that represents values exceeding 255, or use languages that have alphabets larger than 255 letters. This means that we must group collections of bytes together in memory to represent larger data items.

Tagged data

The ASCII standard for representing characters of the (primarily English) alphabet only uses a single byte per character. As these can be represented as simple arrays of bytes we could avoid the problem of larger data groups by using some kind of tagged data or data markup language like XML, JSON, or YAML. These are ASCII encoded text files that tag each data item with a type string and then follow that with a data string representing the data. The programmer can use code libraries that know how to decode these ASCII strings into binary data that is consistent with the CPU architecture. https://zserge.com/jsmn/

If however we want to use binary representations of data larger than a byte we must contend with the following quirks.

Endianness

Some architectures splay larger numerical data into memory starting with the least significant byte of the data stored into an address of memory, followed by the more significant byte of the data stored in the next higher address of memory. These are called "little endian" systems because the little end comes first in memory.

Data	**0xABCD**
Address 0	0xCD
Address 1	0xAB

Endianness

Other architectures splay larger numerical data into memory starting with the most significant byte of the data stored into an address of memory, followed by the least significant byte of the data stored in the next higher address of memory. These are called “big endian” systems because the big end come first in memory.

Data	0xABCD
Address 0	0xAB
Address 1	0xCD

Endianness

The terms little endian and big endian come from Johnathan Swift's Guliver's Travels, a political satire where two factions are at war because one faction opens hard boiled eggs from the big end, while the other faction opens theirs from the little end.

Endianness

If the foreign data endianness does not match the CPUs endianness, the data will appear byte swapped when read in. For us to compensate for this, the foreign data must declare what endianness they used when the data was created. If this does not match the endianness of our CPU we must byte swap that data upon import into our CPU memory. In order to write portable code that will work on other CPU architectures we may use in the future, macros are often used that will either do a byte swap or do nothing depending on the endianness of the CPU.

There isn’t a right way or a wrong way. Depending on the context, sometimes it is advantageous to keep data in big endian format, in other cases little endian is preferred.

Network byte order

One notable case is network sockets programming. Data used in sockets data structures is kept in “network byte order” which is always big endian. The sockets library contains a set of macros;

- htons() Converts a 16-bit unsigned integer (short) from host byte order to network byte order.
- htonl() Converts a 32-bit unsigned integer (long) from host byte order to network byte order.
- ntohs() Converts a 16-bit unsigned integer (short) from network byte order to host byte order.
- ntohl() Converts a 32-bit unsigned integer (long) from network byte order to host byte order.

Data alignment

CPU architectures have constraints on what kinds of memory address can hold what kinds of data. Typically CPUs either prefer or demand that 16 bit wide data must begin on even addresses, and 32 bit wide data begin on addresses divisible by 4. CISC machines like the x86 architectures usually spend lots of transistors and power dissipation on circuitry that can handle situations where this is not the case and will handle the exceptions although at a reduced level of performance.

Data alignment - RISC

RISC machines like ARM typically do not contain this circuitry and will cause a segmentation fault if data is accessed at an illegal address. ARM cores version 7 and above will mitigate this in some cases, this but at reduced performance and without atomic access. The ‘R’ in RISC stands for ‘reduced’ and the ARM only has load (LDR) and store (STR) instructions to move data from memory to register and from register to memory. It can not operate on data directly in memory. LDR moves 32 bits of memory to a register and the address must be divisible by 4. STR moves 32 bits of a register to memory and the address must be divisible by 4. LDRB moves 8 bits of data from any address into a register, zeroing all other bits of the register. STRB moves the low 8 bits of a register to any address. There are LDRH and STRH that do similar operations on 16 bits (half) of the register and will only operate with even memory addresses.

https://developer.arm.com/documentation/ddi0406/c/Application-Level-Architecture/Application-Level-Memory-Model/Alignment-support/Unaligned-data-access-restrictions-in-ARMv7-and-ARMv6?lang=en

Data alignment - padding

This problem of data alignment is usually handled by the compiler. If the programmer defines a data structure containing data of mixed sizes, the compiler will silently add extra padding bytes between items to cause the following item to reside on the correct address type. For example in the structure ;

```
Struct foo {
char a;
short int b;
short int c;
long int d;
} foobar;
```

The compiler will add a hidden pad byte after ‘a’ to force ‘b’ to begin on an even address. Similarly it will add two hidden pad bytes after ‘c’ to force ‘d’ to begin on an address divisible by 4. This makes data structures larger than they would otherwise be.

Data alignment – base address

Additionally the compiler (or its library functions like malloc()) will always allocate storage of data structures such that they begin on an optimal address boundary. If you circumvent that by trying to move data structures into regions created by adding offsets to pointers for example you may create segmentation faults.

Data alignment – naturally aligned

One way to avoid this problem is for the programmer to take care when declaring data structures and make them ‘naturally aligned’ by either re-arranging the data items or adding explicit ‘spare’ data items.

```
Struct foo {
char a;
char spare;
short int b;
short int c;
long int d;
} foobar;
```

Packing

A programmer can ask the compiler to “pack” the data structure by not adding any hidden pad bytes. This can be done in some compilers by using the “__pack” pragma when declaring the structure. As with any pragma this is a non-standard extension the C language. In almost all cases this results in worse performance when accessing the data items.

For those architectures that can’t handle miss-aligned data items the compiler will emit code when accessing the non-aligned members that reads them in using the nearest aligned wider access and then bit shifting that data into the correct resulting position. Note that this only works if you access the members using the dot or arrow syntax. If a non-aligned data member’s address is assigned to a pointer for example, then de-referencing that pointer will cause a segmentation fault.

https://stackoverflow.com/questions/8568432/is-gccs-attribute-packed-pragma-pack-unsafe

Floating point data

Data used to represent floating point values must use a format that is compatible with the CPUs floating point processor. Lucky for us almost all CPUs have adopted the IEEE 754 standard for floating point. This means that the binary data format (except for endianness) is the same for all CPUs.

https://www.geeksforgeeks.org/computer-organization-architecture/ieee-standard-754-floating-point-numbers/

Unicode

Text “string” data can be represented in a many ways. As we have seen the ASCII standard is a simple byte based representation of text and is portable among all CPUS. ASCII can not represent all language ‘code points’ however, so more complex representations were developed, Unicode being widely used. Unicode however is a complicated set of standards, that almost no one completely understands.

https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-excuses

https://www.ascii-code.com/

Unicode -encodings

There are encodings of Unicode that can be represented (encoded) using byte streams (UTF8), 16 bit wide streams (UTF16), and 32 bit wide streams (UTF32). The foreign Unicode data could be in any of these formats, and of either endianness although it is supposed to begin with an endianness identifier (Unicode byte order mark) to help you know if you need to swap or not. The foreign data must declare what Unicode encoding it is using. For the gcc compiler on ARM, it will encode Unicode with 32 bit values (the wchar type). You can use the -fshort-wchar compiler option to switch it to use 16 encoding.

Bitfields

While the C language is standardized, the implementation of many C constructs is purposely left open for compiler writers to implement in any way they choose in order to take advantage of certain architecture features.

From a binary data perspective, bitfields are not portable. It is left to each compiler to decide how the bits in the bitfields are to be laid out. The first bit defined could be the most significant or the least. Furthermore, alignment issues as with structures exist but more so. Compilers can allocate bit groups into bytes as they see fit for optimization.

If the foreign data was created using bitfields, they may or may not match your compiler’s interpretation.

Unions

Unions suffer the same fate in that the compiler writers are free to implement them in various ways, with alignment issues like structures but in some cases even weirder. Unions are not portable from a binary perspective.

Marshaling and serializing

To accommodate all of the various quirks of data representation, programs typically use a technique called marshaling. In marshaling the foreign data is considered a byte stream, where each byte of the stream is interpreted using the rules of the foreign data representation. These bytes are then re-assembled into data structures native to the host CPU using its data representation rules. This usually involves byte shuffling and bit shifting.

Marshaling and serializing

Similarly, when sending or storing host data to a foreign data format the native data is serialized by breaking each native data element down into its constituent bytes and reformatting them into a byte stream.

You may be tempted into thinking that if a foreign data structure happens to match your CPUs native format you can skip doing marshaling or serializing. You should consider the future portability of that decision, as it limits your choices of CPU architectures going forward.