

# **The New C Standard** (Excerpted material)

---

**An Economic and Cultural Commentary**

**Derek M. Jones**

derek@knosof.co.uk

### 3.6

byte  
addressable  
unit

#### byte

addressable unit of data storage large enough to hold any member of the basic character set of the execution environment

#### Commentary

octet

A byte is traditionally thought of as occupying 8 bits. The technical term for a sequence of 8 bits is *octet*. POSIX<sup>[7]</sup> defines a byte as being an octet (i.e., 8 bits). These standards also define the terms *byte* and *character* independently of each other.

character  
single-byte  
CHAR\_BIT  
macro  
char  
hold any mem-  
ber of execution  
character set

There is something of a circularity in the C Standard's definition of byte and character. The macro CHAR\_BIT defines the number of bits in a character type and is required to have a minimum value of 8. The definition of byte deals with data storage, while that for the **char** type deals with type.

This term first appeared in print in 1959.<sup>[3]</sup>

#### Other Languages

The DNA encoding used in Carbon based processors uses three *quytes*<sup>[2]</sup> (each of which has one of the four possible values *A*, *C*, *G*, or *T*) to form a *codon*. The 64 possible codons are used to represent different amino acids, which are used to make proteins, plus a representation for *stop*.

byte  
address unique

NOTE 1 It is possible to express the address of each individual byte of an object uniquely.

#### Commentary

banked  
storage

What is an address? In most cases it is taken to be the value representation held in a pointer object. In some cases a pointer object may not contain all the information needed to calculate an address. For instance, the IAR PICmicro compiler<sup>[6]</sup> provides access to more than 10 different kinds of banked storage. Pointers to this storage can be 1, 2, or 3 bytes in size. On such a host the address of an object is the combination of the pointer value and information on the bank being referred to (which is encoded either in the instruction accessing the byte or a status register).

There is no requirement that the *addressable unit* be the address returned by the address-of operator, with the object as its operand. An implementation may choose to indirectly reference the actual storage used to hold an object's value representation via a lookup table, indexed by the value returned by the address-of, **&**, operator. This index value is then the *address* seen by programs.

Objects are made up of one or more bytes and it is possible to calculate values that point at any of them. The standard is therefore requiring that the individual bytes making up an object are uniquely identifiable. An implementation cannot hide the internal bits of an object. The ordering of bytes within an object and their relative addresses is not defined by the standard (although the relative order of structure members and array elements is defined). This requirement does not prevent more than one object from sharing the same address (i.e., their lifetimes may be different, or they may be members of a union type).

object  
contiguous  
sequence of bytes  
object  
point at each  
bytes of  
value  
copied using  
unsigned char  
structure  
members  
later compare later  
lifetime  
of object  
union type  
overlapping  
members

#### Common Implementations

word addressing

So-called *word addressed* processors only assign unique addresses to units of storage larger than 8 bits (for instance, 32-bit quantities, the word). Implementations for such processors have two choices: (1) define a byte to be the word size (e.g., the Motorola DSP56300 which has a 24 bit word<sup>[8]</sup> and the Analog Devices SHARC has a 32-bit word<sup>[11]</sup>), or (2) addresses of types smaller than a word use a different representation. Unless there is hardware support, the choice is usually to select the first option. With hardware support, some vendors chose option 2 (or to be exact they did not like option 1 and added support for a different representation, the offset of the byte within a word being encoded in additional representation bits). For instance, the Cray PVP<sup>[5]</sup> uses a 64-bit representation for pointers to character and **void** types and a 32-bit value representation for pointers to other types (a 64-bit object representation is used); a version of PRIME Computers C compiler used 48-bit and 32-bit representations, respectively.

For the HP 3000 architecture, it was necessary to be extremely careful of conversions from byte addresses to word addresses and vice versa because of negative stack addressing, which makes byte addresses ambiguous. The original Hewlett-Packard MPE processor (not the later models based on PA-RISC) was word addressed, using 16-bit words (128 K bytes). Byte addresses were signed from -64 K to +64 K, the sign depending on the current setting of the DL and Z registers. This scheme made it possible to access all the storage using byte addresses, but it created an ambiguity in the interpretation of a byte address value (this could only be resolved by knowing the settings of two processor registers).

Some processors are *word addressed* and don't use all the available representation bits. For instance, the Data General Nova<sup>[4]</sup> used only 15 of a possible 16 bits to address storage. The convention of using the additional bit to represent one of the two 8-bit quantities within a 16-bit word was adopted as a software solution to a hardware limitation (i.e., two software routines were provided to read and write individual bytes in storage, using addresses constructed using this convention).

The C Standard recognizes that not all pointer types have the same representation requirements.

alignment  
pointers

### Coding Guidelines

Developers often assume more about the properties of the addresses of bytes than the C Standard requires of an implementation support; for instance, the extent to which bytes are allocated at similar addresses (an important consideration if performing relational operations on those addresses). The extent to which it is possible to make assumptions about the addresses of bytes is discussed elsewhere.

object  
contiguous  
sequence of bytes

### Example

```

1  #include <stdio.h>
2
3  extern short glob;
4
5  void f(void)
6  {
7  int working_g = glob;
8  /*
9   * Write out bytes, starting from least significant byte.
10  */
11  for (int g_index=0; g_index < sizeof(glob); g_index++)
12  {
13  unsigned char uc = (working_g & UCHAR_MAX);
14  fwrite(&uc, 1, 1, stdout);
15  working_g >>= CHAR_BIT;
16  }
17
18  /*
19   * Will the following generate least or most significant
20   * byte orderings? Could be same or different from above.
21  */
22  fwrite(&glob, sizeof(glob), 1, stdout);
23  }

```

55 NOTE 2 A byte is composed of a contiguous sequence of bits, the number of which is implementation-defined.

### Commentary

Implementations where, for instance, the first four bits of a byte are separated from the second four bits by a single bit that is not part of that byte are not permitted. The type **unsigned char** is specified to use all the bits in its representation. The representational issues of this contiguous sequence of bits are discussed elsewhere.

The number of bits must be at least 8, the minimum value of the CHAR\_BIT macro.

value  
copied using  
unsigned char  
value repre-  
sentation  
object repre-  
sentation  
CHAR\_BIT  
macro

### Common Implementations

At the hardware level, storage can use one or more parity bits for error detection and correction (more than one bit is required for this). These additional bits are not visible through the conventional storage access mechanisms. Some hardware platforms do provide methods for accessing the bits in storage at the chip level.

---

low-order bit	The least significant bit is called the <i>low-order bit</i> ;	56
	<b>Commentary</b> This defines the term <i>low-order bit</i> .	
high-order bit	the most significant bit is called the <i>high-order bit</i> .	57
	<b>Commentary</b> This defines the term <i>high-order bit</i> .	

## References

1. Analog Devices, Inc. *C/C++ Compiler and Library Manual for SHARC*. Analog Devices, Inc, 4.0 edition, Jan. 2003.
2. Anon. How genes work: A quick guide to life's operating system. *The Economist*, June 29 2000.
3. W. Buchholz. Origin of the term *byte*. *Annals of the History of Computing*, 3(1):72–72, 1981.
4. Data General Corporation. *Programmer's Reference Manual: Nova Line Computers*. Data General Corporation, 2 edition, Sept. 1975.
5. S. Graphics. *Cray Standard C/C++ Reference Manual*. Silicon Graphics, Inc, Mountain View, CA, USA, 3.6 edition, June 2002.
6. IAR Systems. *PICmicro C Compiler: Programming Guide*, iccpic-1 edition, 1998.
7. ISO. *ISO/IEC 9945-1:1990 Information technology —Portable Operating System Interface (POSIX)*. ISO, 1990.
8. Motorola, Inc. *DSP563CCC Motorola DSP56300 Family Optimizing C Compiler User's Manual*. Motorola, Inc, Austin, TX, USA, 19??