A Character Reference [Updated]

Posted on September 14, 2012

In an earlier posting, portable integer data types were examined in It takes all types… This posting continues this thought with an examination of the character data in realm of embedded systems programming.

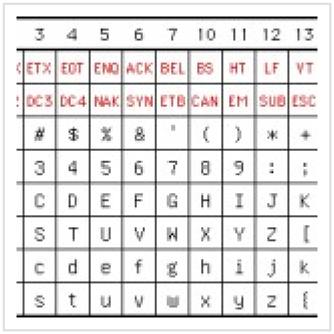
The “C” programming language has supported the character data type since in infancy of the language in 1972. Well sort of a supported data type. In a fashion typical of the extreme minimalism of early “C”, development the “char” data type was made to serve three distinct and often incompatible uses. Char is (was) used for:

- The coding of a single character of data. A pointer to characters became synonymous with a pointer to an array of characters terminated with a null character. This is what passed for strings in”C”.
- The smallest integer data type supported by the processor. This small data type was attractive when the range of values was known to be very limited and space was tight (as it always was)!
- The smallest unit of addressable storage. As such pointers to “char” were treated as interchangeable with pointer of any type. This usage of “char” has been obsolete for quite some time now with pointers to “void” fulfilling this role.

There are two main issues with this state of affairs.

The first is that in “C”, by default all integer data types are signed. For integers this makes sense as this matches the traditional view of integer variables as a subset of integers in mathematics. Thus the integer usage of “char” mandates that it too must default to being signed. This is in direct conflict with “char” for character data. Textual data does not posses a sign and while the meaning of the words may be negative, the text of those words cannot be negative!

Despite common sense, “C” defines “char” as signed. Over the years, many compilers have given the programmer the option to default it to be unsigned instead, but this is hardly universal.



3	4	5	6	7	10	11	12	13
ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT
DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC
#	$	%	&	'	(	)	*	+
3	4	5	6	7	8	9	:	;
C	D	E	F	G	H	I	J	K
S	T	U	V	W	X	Y	Z	[
c	d	e	f	g	h	i	j	k
s	t	u	v	w	x	y	z	{

The second issue that is encountered is the mapping of character glyphs, like ‘C’ to a code like 67. Early in the history of computers, mapping were in chaos. Then the ANSI committee came up with the ASCII code (pictured to the left, labeled in octal) in 1965 and order was restored, briefly!

ASCII is a seven bit code that handles text in English, Numbers plus some punctuation. Soon the demand arose for accented characters, new punctuation and much more. Most processors use eight bit bytes, so most characters had 128 unused codes available for other uses (assuming you got around the signed nonsense) The big problem was that while 128 free codes was enough for a few languages, it was not enough for all of them. Thus was born the Codepage. The Codepage was a number that allowed the selection of one character mapping from many. Now, in embedded systems, Codepages were not generally used. The application hardwired the mapping of code to glyphs. As applications become more sophisticated, this option becomes less attractive.

And then there is the problem of Asian character sets. Here 128 extra codes is quite inadequate. Many thousands of codes are required. Over the years, many strange and bizarre coding schemes have been created to handle Japanese, Chinese, Korean and other Asian languages. I am glad to say that I won’t waste time discussing them as they are all obsolete. They have been swept aside by Unicode!

Unicode defines a huge 1 million code character space and a smaller 65536 subset. These characters may be encoded through a number of schemes; some of these are:

1. UTF-32: this massive four byte format can encode the full 1 million character set. It is seldom used due to it’s space inefficiency and the fact that codes greater than 65535 are seldom required. There are two flavors of UTF-32: Big Endian and Little Endian (see below).
2. UTF-16: this format based on two byte format uses 2 bytes for most glyphs and 4 for seldom used (or supported) ones. There are two flavors of UTF-16: Big Endian and Little Endian (see below).
3. UTF-8: this variable length encoding includes traditional seven bit ASCII. That is, any valid seven bit ASCII string is also a UTF-8 string. In addition, the coding is such that data may be processed one byte at a time. This means that UTF-8 strings can work with most standard “C” library routines. This compatibility is a huge advantage! Further, UTF-8 is the most compact encoding with glyphs ranging from 1 to 4 bytes long or 1 to 3 bytes long for the 65535 code subset. The downside is that the encoding is variable. To find the N’th character in a string, it is necessary to scan the string. It also complicates the allocation of buffers and such.

So what is to be done in an embedded system? I think it is foolish to imagine that one designer can select one coding scheme in advance for all others. It is for that reason that the emSystem software library provides the choice of four possible encodings:

```
//************************************************************
// Advanced character support configuration
//************************************************************
#define CHAR_ASCII_7     0   // Standard 7 bit ASCII.
#define CHAR_ASCII_8     1   // The 8 bit extension to ASCII.
#define CHAR_UTF_8       2   // The var len extension to ASCII.
#define CHAR_UTF_16      3   // The 16 bit extension to ASCII.

// What sorts of characters are to be supported?
#define CHAR_SUPPORT     CHAR_ASCII_7
```

How support for these four options are implemented is the topic of a future posting. For more information on ASCII and Unicode Encoding please visit the ASCII Table. As always, your comments and suggestions are most welcomed.

Peter Camilleri (aka Squidly Jones)

Note: Big Endian vs Little Endian refers to a holy war in computer science about how bytes in multibyte data should be ordered. The Big Endian camp favors the most significant byte being first (in the lowest address) while the Little Endian camp favors the least significant byte being first. To help avoid chaos, a byte order mark may be placed at the start of a text to indicate the byte ordering in use. The names given these byte orderings trace back over 200 years to the Jonathan Swift book Gulliver’s Travels.

[**Update 1**] A now for Unicode the Movie! Starring all 109,242 characters in the Unicode Version 6.0 specification (a cast of 0.1 millions) it is amazing how many forms of written/printed language exist on this planet!

[Update 2] I don’t know how I could have missed adding a link to the Unicode Consortium.

SHARE THIS:

Print LinkedIn Facebook Twitter

LIKE THIS:

Loading...

This entry was posted in **Embedded Development** by **Peter Camilleri**. Bookmark the **permalink [http://teuthida-technologies.com/?p=1291]** .