An Overview of Unicode including ASCII and UTF-8

Harry H. Porter III

HHPorter3@gmail.com

9 February 2022

Abstract

- Unicode is introduced and explained.
- The ASCII character set is listed.
- The UTF-8 encoding is introduced and explained.

Table of Contents

Chapter 1: ASCII

The ASCII Character Set

The ASCII character set is older and simpler than Unicode, so we describe ASCII first.

The ASCII character code is a 7 bit code, in which each code number is assigned to a single character. There are 128 different ASCII codes, over the range:

decimal	hex	binary
0	00	0000 0000
1	01	0000 0001
...	...	...
127	7F	0111 1111

ASCII characters are stored with exactly one character per byte.

Since ASCII is a 7 bit code, the most significant bit of the byte is always 0. In other words, the following byte values are not used in ASCII. This will become important when we discuss UTF-8, which uses these values:

decimal	hex	binary
128	80	1000 0000
129	81	1000 0001
...	...	...
255	FF	1111 1111

The table on the following page lists the ASCII character set, giving the character corresponding to each numerical code.

Most codes correspond to "printable" characters but ASCII also contains some "control characters".

Number of printable characters	84
Number of control characters	34
Total	128

Most of the control characters have only historical significance. They are not widely used and the full table below simply includes them with no description. The more common control characters which you may encounter are:

Decimal	Hex	Description		
0	00	NUL	/0	*Null*
7	07	BEL	/a	*Bell/Alert*
8	08	BS	/b	*Backspace*
9	09	HT	/t	*Tab*
10	0A	LF	/n	*Linefeed/Newline*
13	0D	CR	/r	*Enter/Return*
27	1B	ESC	/e	*Escape*
127	7F	DEL	/d	*Delete*

Note that all the control characters are grouped at the beginning (in the range 0x00 … 0x1F) except for the "delete" character (0x7F) which occurs in the last place.

In the past, ASCII keyboards were not perfectly standardized.

For example, the backspace key on the keyboard may be labeled with "DELETE" or a left arrow or something else; hitting this key may result in the "backspace" BS character (0x08) or the "delete" character (0x7F) or something else being sent to software. Likewise, hitting the key labelled "RETURN" or "ENTER" may result in LF (0x0A) or CR (0x0D) being sent to the software. The Unix/Linux system was able to deal with the variety of keyboards, but at a cost of significant programming complexity.

Modern keyboards are more complex and have much greater flexibility, allowing multi-key combinations, non ASCII characters, etc.

dec	hex				dec	hex		dec	hex			
0	00	NUL	\0	Null	42	2A	*	85	55	U		
1	01	SOH			43	2B	+	86	56	V		
2	02	STX			44	2C	,	87	57	W		
3	03	ETX			45	2D	-	88	58	X		
4	04	EOT			46	2E	.	89	59	Y		
5	05	ENQ			47	2F	/	90	5A	Z		
6	06	ACK			48	30	0	91	5B	[		
7	07	BEL	\a	Bell, alert	49	31	1	92	5C	\		
8	08	BS	\b	Backspace	50	32	2	93	5D	]		
9	09	HT	\t	Tab	51	33	3	94	5E	^		
10	0A	LF	\n	Linefeed/Newline	52	34	4	95	5F	_		
11	0B	VT			53	35	5	96	60	`		
12	0C	FF			54	36	6	97	61	a		
13	0D	CR	\r	Enter/Return	55	37	7	98	62	b		
14	0E	SO			56	38	8	99	63	c		
15	0F	SI			57	39	9	100	64	d		
16	10	DLE			58	3A	:	101	65	e		
17	11	DC1			59	3B	;	102	66	f		
18	12	DC2			60	3C	<	103	67	g		
19	13	DC3			61	3D	=	104	68	h		
20	14	DC4			62	3E	>	105	69	i		
21	15	NAK			63	3F	?	106	6A	j		
22	16	SYN			64	40	@	107	6B	k		
23	17	ETB			65	41	A	108	6C	l		
24	18	CAN			66	42	B	109	6D	m		
25	19	EM			67	43	C	110	6E	n		
26	1A	SUB			68	44	D	111	6F	o		
27	1B	ESC	\e	Escape	69	45	E	112	70	p		
28	1C	FS			70	46	F	113	71	q		
29	1D	GS			71	47	G	114	72	r		
30	1E	RS			72	48	H	115	73	s		
31	1F	US			73	49	I	116	74	t		
32	20	*<space>*			74	4A	J	117	75	u		
33	21	!			75	4B	K	118	76	v		
34	22	"			76	4C	L	119	77	w		
35	23	#			77	4D	M	120	78	x		
36	24	$			78	4E	N	121	79	y		
37	25	%			79	4F	O	122	7A	z		
38	26	&			80	50	P	123	7B	{		
39	27	'			81	51	Q	124	7C	|		
40	28	(			82	52	R	125	7D	}		
41	29	)			83	53	S	126	7E	~		
					84	54	T	127	7F	DEL	\d	Delete

Chapter 2: Unicode

The Unicode Character Set

Throughout the world, there are many characters in use in different languages. The Unicode system is an attempt to capture all the world's characters so they can be represented in computer memory and presented graphically on screens for people to see and read.

The Unicode character set is defined, enumerated, and maintained by a committee. New characters are being added periodically. As of 2020, **Unicode version 13.0** contains **143,859 characters**. The Unicode character set also includes mathematic symbols and emoji.

Each character is assigned

- A number (called a "**codepoint**")
- A glyph (the image)
- A name
- A category

For example:

- Codepoint: 8,713 (= 0x2209)
- Glyph: ∉
- Name: "NOT AN ELEMENT OF"
- Category: Math Symbol

The number of Unicode characters is limited to a maximum of 1,114,112 characters. Roughly 12% of the available "**codepoints**" have been assigned, so there are plenty of unassigned codepoints.

The maximum number of characters is:

decimal	hex
1,114,112	0x11,0000

The codepoints are numbered:

	decimal	hex
min	0	0x00,0000
max	1,114,111	0x10,FFFF

An important fact about Unicode and ASCII is:

> *The entire ASCII character set (printable characters and control characters) is included directly into Unicode. The Unicode codepoint for each character is exactly the same as the ASCII encoding. Thus, ASCII is a proper subset of Unicode.*

Complications and Details

Unicode is more complicated than described in this document. Here, we'll just mention a few of the complexities.

Planes

The Unicode system groups characters into "**planes**". The "**Basic Multilingual Plane**" (BMP), includes the first 65,536 codepoints (0x0000 … 0xFFFF). This plane includes almost every character you'll want to use. In total, there are 17 planes, each of which contains 65,536 codepoints. Most are yet to be filled in.

Each codepoint has a "major category" and a "minor category". For example "∉" has major category "Symbol" and minor category "Math". The character "A" which is called "LATIN CAPITAL LETTER A", has a major category of "Latin" and a minor category of "Upper".

Accent Marks

Unicode includes support for accent characters. In some cases, there is a character with the accent included (as for example, é). But, for characters without such variants, there are special "accent characters", which are intended to apply to the previous character. So, a single "e" would be followed by the accent character.

For example, the following three things are distinct "characters":

Decimal	Hex	Character	Official Unicode Name
101	0065	e	LATIN SMALL LETTER E
180	00B4	´	ACUTE ACCENT
233	00E9	é	LATIN SMALL LETTER E WITH ACUTE

Characters that Look Very Similar

There are a number of characters which may look identical but which are completely different. Below is an example. These character all look identical in "font1", but look different in font2, as I hope you can see.

Decimal	Hex	Font1	Font2	Official Unicode Name
72	48	H	H	LATIN CAPITAL LETTER H
919	0397	H	Η	GREEK CAPITAL LETTER ETA
1053	041D	H	Н	CYRILLIC CAPITAL LETTER EN

Thus, there are multiple ways to encode what is (in some sense) the same character. In some contexts, this presents a security risk, since the user may be spoofed into believing that one identifier is something is not. Programmers beware: equality is not straightforward.

Right-to-Left vs. Left-to-Right

Unicode includes support for languages that are written right-to-left, as well as left-to-right.

Unicode includes support for how and where lines are broken, this is, where newlines are automatically insert into text which spans multiple lines.

The Byte-Order-Mark

Unicode contains something called the "**Byte Order Mark**" (BOM). The BOM is used in conjunction with a similar codepoint, which is declared to be illegal and which must never appear in any Unicode text.

Decimal	Hex	Description
65,279	FEFF	BYTE ORDER MARK
65,534	FFFE	illegal

Note that the above two codepoints are identical if you swap the byte order. A Unicode text may always contain the BOM. Typically the BOM would be the first character in the text, if it is included at all. The BOM prints as an invisible character. Unicode describes this invisibility as "ZERO WIDTH NO-BREAK SPACE".

The Byte Order Mark (BOM) is useful whenever Endianness is an issue. This primarily affects UTF-16 (UTF-16 is less widely used than UTF-8 since UTF-8 seems to be superior.)

If the software encounters a BOM, then everything is okay. On the other hand, if the software encounters the illegal codepoint of 0xFFFE, then it can conclude that it has got the byte order wrong and needs to switch bytes.

Character Classification

Characters fall into classes, such as:

letter
number / digit
mathematical symbol
punctuation
upper case / capital
lower case
space
white-space

With so many different languages and characters, these tests should not be done by hand, as was possible in the ASCII system. Instead, functions should be used, in order to encapsulate and hide the details of Unicode. And presumably these functions will need to be updated and modified, as Unicode evolves and new versions are released.

Alphabetization and Ordering

It is often required to alphabetize words. In English, this is straightforward for anyone who has learned the alphabet. The key operation needed to sort a list is being able to compute a < relationship between two strings. When the strings are Unicode texts — and may contain characters from many languages — any definition of “alphabetic order” is more complex.

The Replacement Character

One unusual character is the “**replacement character**”, shown below. This character glyph (i.e., this graphic image) is supposed to be substituted by fonts that do not contain a character. So when the software encounters a codepoint which is defined by Unicode but which is not present in the font, the “replacement character” is to be used.

Decimal	Hex	Glyph[1]	Official Unicode Name
65533	FFFD		REPLACEMENT CHARACTER

If you see the image of the replacement character in printed text, it indicates that some other character is present but the software is incapable of rendering that character.

[1] The software I am using to create this document — Apple’s “Pages” — treats the replacement character differently from other characters and refuses to display it. Thus, I was forced to include an image of the glyph.

Chapter 3: UTF-8

Character Encoding

There are several ways to encode a Unicode character or string of Unicode characters.

The **UTF-32 encoding** simply uses a word (= 4 bytes = 32 bits) to encode each codepoint. This encoding is good for encoding individual characters, but is very wasteful for long strings. Thus, UTF-32 is not widely used for Unicode strings.

The **UTF-8 encoding** is widely used and will be discussed in detail the following section.

The **UTF-16** encoding is not as widely used and will not be discussed here.

Another encoding is meant to be human readable. For example the "∉" character is encoded as:

    U+2209

The prefix "U+" is followed by hex characters giving the numerical codepoint. Generally speaking, there will be exactly 4 hex characters. But since Unicode contains some codepoints greater than 0xFFFF, 4 hex characters will not always be enough. There are different approaches to dealing with this. One common approach is to follow the "U+" prefix by either 4 or 6 hex digits.

The Python language allows the user to write Unicode characters within strings in several ways as shown in these examples. (These all produce the same string.)

```
" d ∉ {a,b,c} "
" d \u2209 {a,b,c} "
" d \U00002209 {a,b,c} "
" d \N{NOT AN ELEMENT OF} {a,b,c} "
```

In Python, strings are encoded using UTF-8. Thus, the following will not work:

```
" d \x22\x09 {a,b,c} "
" d \x00\x00\x22\x09 {a,b,c} "
```

The UTF-8 Encoding

As mentioned above, one approach to encoding Unicode strings is to use 4 bytes per character, but this is wasteful of space. The UTF-8 encoding scheme is variable length. Each character is encoded with between 1 and 4 bytes. Common characters tend to have shorter encodings.

Since Unicode is limited to 1,114,112 codepoints, the largest code point is:

decimal	hex	binary
1,114,111	10,FFFF	1 0000 1111 1111 1111 1111

As you can see, at most 21 bits are needed for each codepoint. However, since the leading bits of many common codepoints are zero, the UTF-8 can use fewer bits for many codepoints.

Depending on the value of the codepoint, a different number of bytes is used.

1 byte is used for codepoints in this range:

decimal	hex	binary
0	0	000 0000
...	...	...
127	7F	111 1111

2 bytes are used for codepoints in this range:

decimal	hex	binary
128	80	000 1000 0000
...	...	...
2,047	7FF	111 1111 1111

3 bytes are used for codepoints in this range:

decimal	**hex**	**binary**
2,048	800	0000 1000 0000 0000
...	...	...
65,535	FFFF	1111 1111 1111 1111

4 bytes are used for codepoints in this range:

decimal	**hex**	**binary**
65,536	1,0000	0 0001 0000 0000 0000 0000
...	...	...
1,114,111	10,FFFF	1 0000 1111 1111 1111 1111

Next, we give the UTF-8 encoding scheme. In the following, `xxx…xxx` is the binary form of the codepoint. We can refer to these bits as the "**payload**".

Frankly, I can't describe UTF-8 more concisely and clearly than the following image, which is from Wikipedia.

Number of bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	0xxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

UTF-8 Encoding Examples

First, consider the following character:

Decimal	Hex	Character	Official Unicode Name
97	61	a	LATIN SMALL LETTER A

Since this codepoint is an ASCII character, it is encoded in one byte, exactly as is:

01100001	*Binary encoding*
0x61	*(in hex)*

Next, consider the following character:

Decimal	Hex	Character	Official Unicode Name
233	00E9	é	LATIN SMALL LETTER E WITH ACUTE

Since this codepoint is in the range requiring a two-byte encoding, it is encoded as follows:

Codepoint U+00E9:	0 0000 0000 0000 1110 1001
Regrouping the bits:	00011 101001

Header	Extension	
110-----	10------	*Encoding template*
00011	101001	*Payload*
11000011	10101001	*Complete encoding*
0xC3	0xA9	*(in hex)*

Finally, consider this character:

Decimal	Hex	Character	Official Unicode Name
2,322	0912	ऒ	DEVANAGARI LETTER SHORT O

Since this codepoint is in the range requiring a three-byte encoding, it is encoded as follows:

Codepoint U+0912:	`0 0000 0000 1001 0001 0010`
Regrouping the bits:	`0000 100100 010010`

Header	Extension	Extension	
`1110----`	`10------`	`10------`	*Encoding template*
`0000`	`100100`	`010010`	*Payload*
`11100000`	`10100100`	`10010010`	*Complete encoding*
`0xE0`	`0xA4`	`0x92`	*(in hex)*

UTF-8 and ASCII Text Files

The UTF-8 encoding has the following important property:

> *Any string of characters that contains only ASCII characters and ASCII control characters is represented identically in UTF-8. A text file containing only legal ASCII characters is indistinguishable from a UTF-8 file which contains only ASCII characters; there is no difference in the encodings, if only ASCII characters are present in the strings.*

This means that any software that handles UTF-8 strings can be given an ASCII encoded string and it will perform correctly.

Also, any legacy software that expects ASCII encoded strings and that deals with bytes outside the ASCII range (i.e., 0x80 … 0xFF) by printing these bytes using escapes (or ignoring them altogether) will work reasonably well if accidentally given a UTF-8 encoded string. For example, the valid ASCII characters will be printed correctly, and the non-ASCII character will print using escape codes.

In particular, control code like \n (newline) and \0 (null) will work exactly the same in either UTF-8 and ASCII.

Determining the "**string length**" of an ASCII string is straightforward and unambiguous. The number of characters and the number of bytes will always be identical. With a UTF-8 string, "length" can mean either;

- The number of bytes
- The number of characters.

Accessing a character using an integer index in an ASCII string is straightforward. For example:

str[4000] *Retrieve a character from a string*

Since a string of ASCII characters is an array of bytes, this operation is fast.

With a UTF-8 encoded string, locating the a character by index requires a lot of time, since the string must be scanned character-by-character. (More precisely, the operation is linear in the magnitude of the index.)

Modifying a character within an ASCII string is straightforward: a single byte is replaced with another value. However, with a UTF-8 string we have a problem since the character being replaced may be a different size than the new character. As a result, we may have to insert additional bytes or remove existing bytes. As a result, the length of the string in bytes may change. With long strings, this may require significant amounts of copying.

UTF-8 Error Conditions

Not all byte sequences are legal UTF-8 strings. It is possible that a binary file, when analyzed as a UTF-8 encoded Unicode string, will contain errors.

Error 1: Invalid Byte Prefix

We can view a multi-byte UTF-8 encoded character as consisting of a “header byte”, followed by 1-3 “extension bytes”.

All UTF-8 bytes begin in one of the following ways:

`0-------`	ASCII character
`10------`	Extension byte
`110-----`	Header byte
`1110----`	Header byte
`11110---`	Header byte

Any byte that begins as follows is illegal:

`11111---`	Illegal bytes

Error 2: Missing Extension Byte

The header byte indicates how many extension bytes will follow it.

`110-----`	Header byte; followed by 1 extension byte
`1110----`	Header byte; followed by 2 extension byte
`11110---`	Header byte; followed by 3 extension byte

If the header byte is not followed by the required number of extension bytes, it is an error. In other words, if one or more extension bytes is missing, it is an error.

Error 3: Unexpected Extension Byte

A related error is having too many extension bytes.

Extension bytes may only follow header bytes. Each header byte must be followed by exactly the number of extension bytes expected. An extra extension byte is an error. Furthermore, any extension byte that appears in isolation is in error.

Error 4: Wrong Encoding

The UTF-8 encoding scheme is based on ranges. For example, a codepoint in the range U+0080 … U+07FF is supposed to be encoded with 2 bytes. For example U+0321 is supposed to be encoded as:

```
Codepoint U+0321:        0000 0011 0010 0001
Regrouping the bits:            01100 100001

  Header      Extension
110-----      10------        Encoding template
   01100        100001        Payload
11001100      10100001        Complete encoding
  0xCC          0xA1          (in hex)
```

However, if the codepoint is encoded with more bytes than required, it is an error. The following is an encoding of the same value (U+0321), but this encoding is illegal:

```
Codepoint U+0321:        0000 0011 0010 0001
Regrouping the bits:     0000 001100 100001

  Header     Extension   Extension
1110----     10------    10------        Encoding template
    0000       001100      100001        Payload
11100000     10001100    10100001        Complete encoding
  0xE0         0x8C        0xA1          (in hex)
```

It seems reasonable for software to ignore this error and to tolerate any such incorrectly encoded characters.

Error 5: Undefined Codepoint

The Unicode system can accommodate up to 1,114,111 codepoints. However, as of this writing, the Unicode standard defines only 143,859 codepoints.

decimal	hex	
1,114,111	10,FFFF	*Maximum codepoint in the future*
143,858	02,31F2	*Largest defined codepoint to date*
344,865	05,4321	*An undefined codepoint*

An undefined codepoint should never appear in a UTF-8 string. (Note that this condition implicitly disallows any codepoint greater than 0x10,FFFF.)

For example, the string containing the character U+054321 would be illegal since it specifies an undefined character. Here is the UTF-8 encoding for this undefined codepoint:

Codepoint U+054321:	0 0101 0100 0011 0010 0001
Regrouping the bits:	001 010100 001100 100001

Header	Extension	Extension	Extension	
11110---	10------	10------	10------	*Encoding template*
001	010100	001100	100001	*Payload*
11110001	10010100	10001100	10100001	*Complete encoding*
0xF1	0x94	0x8C	0xA1	*(in hex)*

Appendix 1: About this Document

Document Revision History / Permission to Copy

Version numbers are not used to identify revisions to this document. Instead the date and the author's name is used. The document history is:

Date	**Author**
4 October 2020	Harry H. Porter III <document created>
9 February 2022	Harry H. Porter III <minor correction>

In the spirit of the open-source and free software movements, the author grants permission to freely copy and/or modify this document, with the following requirement:

> ***You must not alter this section, except to add to the revision history. You must append your date/name to the revision history.***

Any material lifted should be referenced.