Alejandro Ramírez Aldariz

Consultor tecnológico de desarrollo de proyectos informáticos.

Puedes encontrarme en Autentia: Ofrecemos servicios de soporte a desarrollo, factoría y formación

Somos expertos en Java/J2EE

Ver todos los tutoriales del autor

Fecha de publicación del tutorial: 2015-02-24

Índice

Introducción

Este artículo explica como representa texto el ordenador. Desde porque los ordenadores usan el sistema binario, qué es ASCII, a conceptos generales de Unicode. Unicode es un estandar complejo, pero aquí tienes la cultura general necesaria para hablar del tema, y orientarte por ti mismo.

El sistema binario

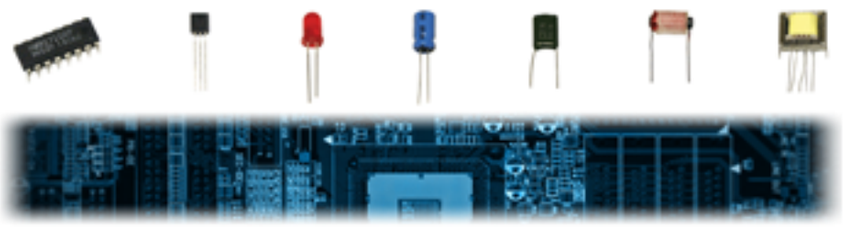
Un **componente electrónico** es un dispositivo capaz de manipular electrones o sus campos asociados. Por ejemplo, transistores, diodos, condensadores, etc.

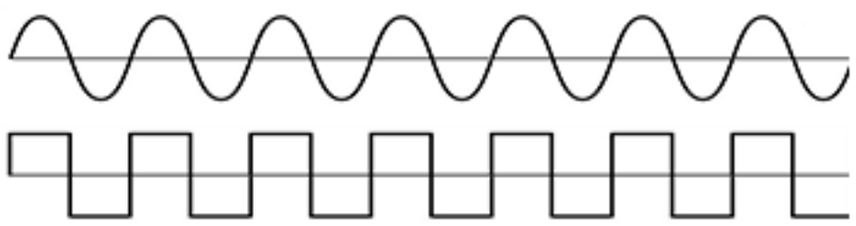
Los componentes electrónicos se combinan y conectan mediante cables para formar **circuitos electrónicos** capaces de realizar operaciones complejas. Por ejemplo, una radio es un circuito electrónico que transforma una señal electromagnética en una señal sonora.

A un circuito electrónico lo llamamos **analógico** si opera con rangos continuos de señales electricas. Por ejemplo, un

altavoz amplifica la señal a un número indefinido de valores entre un volumen mínimo y máximo.

En cambio, si opera con señales discretas, decimos que el circuito electrónico es **digital**. Por ejemplo, un ordenador opera con dos voltajes de valores exactos: cero voltios, y el número de voltios que proporcione la fuente de voltaje, por ejemplo 1.4 voltios. Esto es así porque están construidos con **transistores**, que son componentes electrónicos cuyo funcionamiento es más fiable cuando representa solo dos estados. A grandes rangos, cuantos más estados representas con un rango finito de voltaje, más posibilidades existen de confundir un estado con el otro.





En resumen, los ordenadores expresan información usando solo dos estados porque así sus componentes son más fiables.

El sistema de numeración más fácil de implementar con dos estados es el **sistema numérico binario**. El binario es similar al decimal, pero solo utiliza los digitos uno y cero. Por ejemplo:

decimal	0	1	2	3	4	...
binario	0	1	10	11	100	...

Bits y bytes

Cuando transferimos texto usando un ordenador, existen componentes electrónicos y cables que leen y escriben estas cadenas de ceros y unos. Cada uno de los digitos 0 o 1 que circulan por el ordenador, se considera la unidad mínima de información en informática, y se conoce como **bit**.

Para acelerar la transferencia, los bits se transfieren en grupos de 8, 16, 32, o 64. Cuantos más bits lees a la vez, más cables y componentes necesitas para transferirlos. El equilibrio entre miniaturización, coste de construcción, y otras consideraciones prácticas, hace que los ordenadores actuales operen con bloques de solo 64 bits. Sin embargo hace 50 años, el límite práctico estaba en 8. Un grupo de 8 bits se conoce como **byte**.

El ordenador usa el sistema de numeración binario tanto para expresar datos, como para referenciar esos datos por su posición en memoria. Este es el motivo por el que la información se transfiere en multiplos de dos. ¿Sabes porqué? Es porque la capacidad expresiva de una base de numeración es $base^{\text{número de digitos}}$. Por ejemplo con un digito decimal puedes expresar 10 valores (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), por tanto lo más eficiente es manejar información en multiplos de diez, de otro modo desperdicias la capacidad expresiva de un dígito decimal.

El motivo por el que además son multiplos de 8 es historico. Han existido computadores en los que un byte tenía 1, 6, 7, 8, 9, 12, 18, y 20 bits, pero la popularidad del chip 8080 (el cual usaba 8-bit) hizo que byte fuera sinónimo de 8 bits.

¿Qué es un caracter?

Un **caracter** es una unidad de información abstracta. Algunos ejemplos de caracteres son los números y letras del alfabeto, los signos de puntuación, los emojis, y los caracteres de control como un salto de línea.

Un **juego de caracteres** es una colección de caracteres usada para escribir en un lenguaje particular. Algunos idiomas usan uno, otros usan varios (por ejemplo, kanji y kana del japones), y otros usan variantes de un mismo juego de caracteres (los lenguajes europeos usan variantes del alfabeto romano).

Un **glifo** es una representación visual de un caracter. En un ordenador, los glifos asociados a caracteres suelen estar almacenados en **fuentes**, también llamados tipos de letra. Cada fuente contiene un estilo homogeneo de glifos para un alfabeto. Este es un ejemplo de varios glifos que representan un mismo caracter:

Tipos de caracteres

A grandes rasgos hay tres sistemas de escritura

- **Alfabético**. Un alfabeto es el conjunto de letras usadas para escribir un lenguaje. Cada una representa un fonema o se usa junto a otras para representar un fonema. La mayoría de alfabetos en uso se basan en el alfabeto romano.
- **Silábico**. Contiene caracteres que representan cada una de las sílabas de un lenguaje. Una sílaba es una combinación de vocal o vocal y consonante. Es apropiado para lenguajes con pocas silabas, como el japones, que contiene un centenar. El ingés en cambio, contiene miles de combinaciones de conjuntos de vocales y consonantes.
- **Ideográfico**. Contiene caracteres que representan ideas. Por ejemplo, el chino, o el egipcio antiguo.

Estas categorías no son rígidas. Por ejemplo, el español es alfabético, pero también usa un ideograma como el símbolo de euro, que representa la estabilidad de Europa (es una e de Europa con dos líneas indican estabilidad).

ASCII

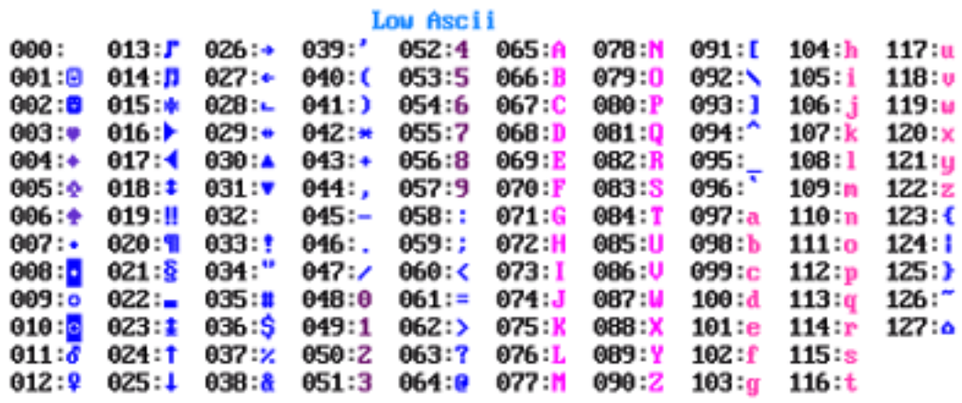
¿Qué es ASCII?

El único tipo de dato que un ordenador es capaz de almacenar y manipular es el número binario. El truco para representar texto es asignar un número a cada letra del alfabeto, e implementar un sistema de renderizado que visualiza el caracter correspondiente a cada número.

Llamamos **código** al sistema de normas para convertir información simple en información codificada. Una **codificación de caracteres** por ejemplo, es un código que convierte caracteres a números.

La codificación de caracteres más popular se llama **ASCII** (American Standard Code for Information Interchange). ASCII asigna números decimales del 0 al 127 a las letras del alfabeto inglés, los números, y algunos signos de puntuación y de control. Los signos de control son indicadores para procesar información, por ejemplo, el salto de línea. La letra “A” en ASCII tiene asignado el número 65 (`0x41` en hexadecimal), y las letras consecutivas del alfabeto tienen asignados números consecutivos:

letra	binario	decimal
A	1000001	65
B	1000010	66
C	1000011	67
D	1000100	68
etc.		



Según este código, la representación de las letras “abc” en binario es la siguiente:

```
1000001 1000010 1000011
```

El documento que estas leyendo en este momento tiene una representación en binario en tu ordenador en forma de ceros y unos. A nivel físico se representa en la memoria mediante la presencia o ausencia de voltaje, y en el disco mediante magnetismo.

Cuando ASCII se publicó en 1964, las máquinas de entonces transferian 8 bits a la vez. Sin embargo ASCII usó solo 7, y reservó el octavo bit para detectar errores de transmisión. Esos 7 bits permiten expresar 2^7 = 128 valores, que ASCII uso para codificar 33 señales de control (usadas entre máquinas), y 95 números, símbolos de puntuación, y letras del alfabeto inglés.

Variantes del ASCII

En los ’80 aparecieron variantes de ASCII que usaban el octavo bit para representar caracteres adicionales, lo que permitio representar caracteres adicionales propios de alfabetos no ingleses, como por ejemplo el español.

Debido a que 8 bits no son suficientes para representar todos los alfabetos del mundo, continuaron apareciendo variantes ASCII de 8 bits incompatibles entre sí. Estas variantes se llaman a veces **ASCII extendido**, pero no forman parte del estandar ANSI.

Hay varios conjuntos de “ASCII extendido”, cada uno de los cuales contiene codificaciones para muchos lenguajes:

- **Windows code pages**, usado en aplicaciones gráficas Windows.
- **OEM code pages**, usando en aplicaciones de consola Windows.
- **ISO-8859** es un estandar ISO para codificación en 8 bits. Tiene 16 partes. La primera se llama ISO-8859-1, también conocida como Latin-1, que cubre la mayoría de lenguajes de Europa occidental.

En ASCII solo es posible trabajar con un alfabeto a la vez. ASCII tampoco es válido para representar alfabetos asiáticos, porque contienen miles de caracteres. Estas deficiencias, junto a la popularidad de la informática, y la potencia creciente de los ordenadores, hicieron deseable y posible la creación de un estandar más exhaustivo.

Unicode

¿Qué es Unicode?

Unicode es una codificación de caracteres que asigna un número a cada uno de los caracteres de practicamente todos los alfabetos existentes, incluyendo las lenguas muertas como el egipcio antiguo y otros. El estandar Unicode lo publica y mantiene el consorcio Unicode, una entidad sin animo de lucro formada por empresas e individuos.

El estandar Unicode también especifica algoritmos sobre como manejar texto, que probablemente solo necesites si realizas tareas especializadas. Por ejemplo:

- Dividir palabras y líneas.
- Ordenar texto.
- Formatear números, fechas, y horas.
- Mostrar texto que fluye de derecha a izquierda.
- Mostrar texto cuya forma escrita se combina y reordena.
- Tratar problemas de seguridad relativos a caracteres parecidos.

Unidades y puntos de código

Unicode comenzó como un sistema de codificación de 16 bits, aunque desde Unicode 2.0 (publicado en 1996), es una codificación de 21 bits.

Un **punto de código** es el número con el que se identifica un caracter en el estandar Unicode. El punto de código se escribe con el formato `U+xxxx` donde las `xxxx` son de cuatro a seis digitos en sistema de numeración hexadecimal.

En Unicode 7.0 el rango válido de puntos de código va de 0 a $10FFFF_{16}$. El hexadecimal se usa por conveniencia en lugar del binario, porque es más fácil recordar. Por ejemplo, Unicode asigna el número 65 a la letra a latina mayúscula. El punto de código correspondiente es `U+0041` porque $65_{decimal} = 0x41_{hexadecimal}$.

Unicode es compatible con la codificación ISO-8859-1, porque los 256 primeros caracteres de Unicode coinciden con los caracteres de ISO-8859-1. Esto hace que la mayoría de texto en uso requiera solo un byte por caracter.

El punto de código se representa con grupos de 8, 16, o 32 bits dependiendo respectivamente, de si el tipo de codificación es UTF-8, UTF-16, o UTF-32. Cada uno de estos grupos se llama unidad de código. Una **unidad de código** es el mínimo grupo de bits necesario para representar una unidad de texto codificado. En UTF-8 es 8 bit, en UTF-16 es 16 bit, y en UTF-32 es 32 bit. Por ejemplo, `U+0041` requiere 2 bytes, es decir, dos unidades de código en UTF-8, y una unidad de código en UTF-16.

¿Como escribir puntos de código en Mac?

Para teclear caracteres en Mac usando puntos de código sigue estos pasos:

- Ve a System Preferences > Keyboard > Input Sources
-
  - Añade Unicode Hex Input.
  - Activa Show Input menu in menu bar.
- Activa el teclado Unicode en la barra de menu.
- Para probarlo, deja pulsado alt y pulsa el código numérico del punto de código que desees (por ejemplo el 0041 para

la letra A).

Unicode no dice nada sobre como debe representarse cada caracter. Por ejemplo, cada uno de los tipos de letra instalados en tu ordenador contiene una representación visualmente diferente de la letra latina A.

Estructura de Unicode

Planos

Unicode se divide en **17 planos**, que son grupos con 65,536 caracteres cada uno. Por tanto los caracteres representables son 17* (2^{16}) = 1,114,112. Sin emabargo, de esos 17 planos, solo 6 tienen caracteres asignados. Y en esos planos, solo hay 112,956 caracteres asignados en Unicode 7.0.

La mayoría de caracteres usados en lenguas occidentales están en el plano 0, que se conoce como **Plano Multilingue Básico**, también conocido por su acrónimo inglés “BMP”.

Plano	Rango	Nombre
Plano 0	`0x0000` a `0xFFFF`	Plano Multilingue Básico (BMP)
Plano 1	`0x10000` a `0x1FFFF`	Plano Multilingue Suplementario
Plano 2	`0x20000` a `0x2FFFF`	Plano Suplementario Ideográfico
Plano 3 a 13	no asignado	
Plano 14	`0xE0000` a `0xEFFFF`	Plano Suplementario de Proposito Especial
Plano 15	`0x0F0000` a `0x10FFFF`	Plano Suplementario de Uso Privado

En el BMP, los 256 primeros caracteres de Unicode se representan con un byte, mientras que el resto de caracteres necesita dos bytes, porque es el número de bits necesario para abarcar el rango de valores del primer plano (2^{16} = 65536).

Bloques

- Un **bloque** es un rango continuo de puntos de código.
- Cada bloque tiene un nombre único. Por ejemplo “Basic Latin”, “Hebreo”, y otros.
- Cada punto de código tiene una propiedad "Block name" que indica el bloque al que pertenece.
- Los bloques de código pueden consultarse en estos enlaces:
  - Unicode Character Code Charts muestra nombres de bloques, nombres de caracteres, y puntos de código
  - Unicode/Character reference muestra rangos y planos
  - Unicode block (wikipedia)

Algunos bloques curiosos

> *“¿Cual es tu bloque favorito de Unicode?”*
> *—dijo ella mientras secaba su larga cabellera con una toalla.*

Jeroglifos egipcios

Unicode soporta jeroglifos egipcios desde Unicode 5.2 con rango `13000–13FFF`. Para mostrar los glifos correspondientes a estos puntos de código necesitas una de las tres fuentes que los soportan. Sin ellas, solo veras cuadraditos en tu navegador. Estas son las fuentes:

- Aegyptus
- New Gardiner
- Noto Sans Egyptian Hieroglyphs

El modo más popular de escribir en egipcio es mediante transliteraciones. La **transliteración** es la expresión de un signo de un lenguaje con un signo de otro. Para convertir una transliteración en su correspondiente punto de có necesitas un software especializado. Las transliteraciones no son parte del estandar Unicode, solo una herramienta usada para escribir jeroglifos. He aqu&iactute; un ejemplo de transliteración y sus caracteres equivalentes en Unicode:

> `ḥtp-ḏỉ-nśwt wśỉr ḫnty ỉmntjw ncr ḏɪ nb ɪbc̣w wp-wɪwt nb tɪ c̣śr`
>
> 

Esto significa, por supuesto,

> oferta real a Osiris, el más importante de los occidentales, el Gran Dios, Señor de Abydos; y de Wepwawet, Señor de la Tierra Sagrada

Otro modo es usar el teclado Unicode, una tabla de referencia, y traducir uno a uno.

Emoji

- Los **emoji** son pictogramas (caracteres que representan imagenes).
- La palabra emoji viene del Japones 絵 (e ≅ imagen) 文 (mo ≅ escritura) 字 (ji ≅ caracter).
- Los emoji representan cosas variadas, como emociones, actividades, banderas, vehículos, comida, etc.
- Un **emoticono** (del inglés emoción e icono) es un modo de mostrar una emoción mediante caracteres. Algunos de los emoji son emoticonos porque representan emociones.
- Los emoji están esparcidos en varios bloques. Las expresiones faciales por ejemplo, están en el rango de puntos de código `1F600–1F64F`.
- En plataformas Apple, los glifos para emoji están en la fuente “Apple Color Emoji”.

Para mostrar la paleta de caracteres Emoji en OS X pulsa dos veces ^⌘ + espacio. Si pulsas botón derecho sobre un emoji y pulsas “Copy Character Info”, aparece la siguiente información:

```
😀 GRINNING FACE
Unicode: U+1F600 (U+D83D U+DE00), UTF-8: F0 9F 98 80
```

¿Qué quieren decir cuando dicen…?

Carácter compuesto

Un **carácter compuesto** es una entidad Unicode que puede definirse como una secuencia de otros caracteres.

- Por ejemplo, `U+00E9` (letra minúscula latina e con acento agudo) tiene el mismo significado y apariencia que `U+0065 U+0301` (letra minúscula latina e, y acento agudo de combinación). Se consideran **canónicamente equivalentes**, pero no iguales porque están hechos de diferentes puntos de código.
- Los caracteres compuestos existen para permitir implementaciones de Unicode en sistemas que solo permiten representar cada caracter por separado.

Los **caracteres compatibles equivalentes** son aquellos que representan el mismo caracter abstracto, pero tienen diferentes apariencias visuales. Por ejemplo, el carácter ﬀ (ligadura latina minúscula `ff`, `U+FB00`) es compatible con (pero no canónicamente equivalente a) la secuencia `ff` (dos letras latinas minúsculas f, `U+0066 U+0066`).

Si queremos comparar texto, por ejemplo durante una busqueda, quizas queramos tratar caracteres canónicamente equivalentes, y caracteres compatibles equivalentes como un mismo caracter. En ese caso necesitaremos aplicar un **algoritmo de normalización** que convierten una cadena a una representación única que puede ser comparada binariamente. Hay cuatro formas de normalización llamadas C, D, KD, y KC. Estos algoritmos permiten comparar caracteres, pero pueden eliminar distinciones de formato que evitan conversiones en ambos sentidos.

Grafema

Un **grafema** es una unidad atómica en un lenguaje.

Un **cluster de grafemas** es una secuencia de puntos de código que representa un grafema.

Glifos

Un **glifo** es un carácter gráfico que representa un caracter abstracto.

- Una **forma variante** es cada uno de los glifos de un conjunto de glifos que representan el mismo caracter abstracto.
- Una **secuencia de variaciones** es un mecanismo para escoger una forma variante. Consiste de un caracter base seguido de uno de los 256 **selectores de variación**, que son `VS1` a `VS256`, `U+FE00` a `U+FE0F`, y `U+E0100` a `U+E01EF`).

Por ejemplo, los glifos ☔ y ☂ representan el caracter abstracto “Paraguas con las gotas de agua”, y son por tanto formas variantes de dicho caracter. La secuencia de variaciones para representarlos son:

- `U+2614 U+FE0F` para ☔
- `U+2614 U+FE0E` para ☂

Caracter percibido por el usuario

Lo que el usuario percibe como caracter. Por ejemplo, en inglés “ch” son dos letras, pero en checo y eslovaco se considera una.

Formatos de transformación Unicode

Los formatos de transformación Unicode (UTF) especifican como serializar un punto de código abstracto a bytes en memoria. Se les conoce popularmente como “codificaciones”, aunque este nombre es incorrecto.

Debido a que Unicode es una codificación que asigna números expresables con 21 bits, y que los ordenadores transfieren datos en multiplos de 8 bits (8, 16, 32, …), hay tres posibles modos de expresar Unicode:

- Usando una unidad de código de 32 bits (UTF-32).
- Usando una o dos unidades de código de 16 bits (UTF-16).
- Usando de una a cuatro unidades de código de 8 bits (UTF-8).

UTF-32

- Formato con unidades de código de 32 bits.
- Representa cada punto de código con exactamente una unidad de código.
- Se usa poco porque los puntos de código más usados necesitan mucho menos de 32 bits.

UTF-16

- Formato con unidades de código de 16 bits.
- Usa una unidad de código para representar puntos de código del BMP y dos para el resto. Al par de unidades de código usadas para representar un punto de código se les conoce como **par sustituto** (surrogate pair).
- El orden de los bytes es normalmente el de la CPU en la que se ejecuta una implementación dada.

UTF-8

Características de UTF-8:

- Formato con unidades de código de 8 bits.
- Usa una unidad de código para los primeros 256 caracteres, dos para los 1920 siguientes, y tres para el resto de puntos de código del BMP. Usa cuatro unidades de código para codificar los puntos de código de otros planos.
- Cuando el texto utiliza unicamente caracteres ISO-8859-1, no hay diferencia entre un texto codificado en ISO-8859-1 y UTF-8.
- UTF-8 es con diferencia el formato más popular debido a su compatibilidad con ASCII e ISO-8859-1, y a su soporte en librerías.

Punto de código a UTF-16

El BMP es un plano de 16 bits con potencial para codificar 2^{16} = 65536 caracteres en puntos de código `U+0000` a `U+FFFF`. Dentro de este rango, los 2048 puntos de código que van de `U+D800` a `U+DFFF` no tienen caracter asignado, y se consideran invalidos.

El algoritmo para convertir un punto de código a UTF-16 es este:

- Si es un punto de código valido del BMP se traduce directamente a UTF-16 usando una unidad de código.
- Si es un punto de código fuera del BMP se procede como sigue:
- ◦ Resta `0x10000` del punto de código (`0x10000` es el rango del BMP).

- Expresa el resultado como binario de 20 bits (rellena con ceros si hace falta).
- Usa los patrones `110110xxxxxxxxxx` y `110111xxxxxxxxxx` para los 10 bits más y menos significativos del resultado del punto anterior.

Por ejemplo, dado el caracter “grinning face” (😀) con Unicode `U+1F600`, su expresión en UTF-16 es `U+D83D U+DE00` porque:

```
0x1F600 - 0x10000 = 0b 0000111101 1000000000
0b1101100000111101 to hex = 0xD83D
0b1101111000000000 to hex = 0xDE00
```

Punto de código a UTF-8

Dependiendo del rango del punto de código a codificar, usaremos uno, dos, tres, o cuatro unidades de código UTF-8.

`0000–007F`	`0xxxxxxx`	7 bits en 1 byte
`0080–07FF`	`110xxxxx, 10xxxxxx`	5+6 bits = 11 bits en 2 bytes
`0800–FFFF`	`1110xxxx, 10xxxxxx, 10xxxxxx`	4+6+6 bits = 16 bits en 3 bytes
`10000–1FFFF`	`11110xxx, 10xxxxxx, 10xxxxxx, 10xxxxxx`	3+6+6+6 bits = 21 bits en 4 bytes

Observa que si el bit más significativo no es cero, el número de unos antes del primer cero indica cuantas unidades de código codifican el punto de código.

Por ejemplo, dado el caracter “grinning face” (😀) con Unicode `U+1F600`, su expresión en UTF-8 es `0xF0 0x9F 0x98 0x83` porque

```
0x1F600 to binary = 0b11111011000000000
     000   011111   011000   000000
11110000 10011111 10011000 10000000
    0xF0     0x9F     0x98     0x80
```

¿Qué UTF usa este fichero?

El modo algoritmico de identificar los puntos de código de un texto es teniendo en cuenta las secciones anteriores, pero el modo rápido que yo uso es este otro.

Cargalo en un editor cualquiera (Textmate, BBEdit, TextWrangler, SublimeText, Pages, Xcode, …) y que te lo diga el editor. —este era mi gran secreto, ahora tú también lo conoces.

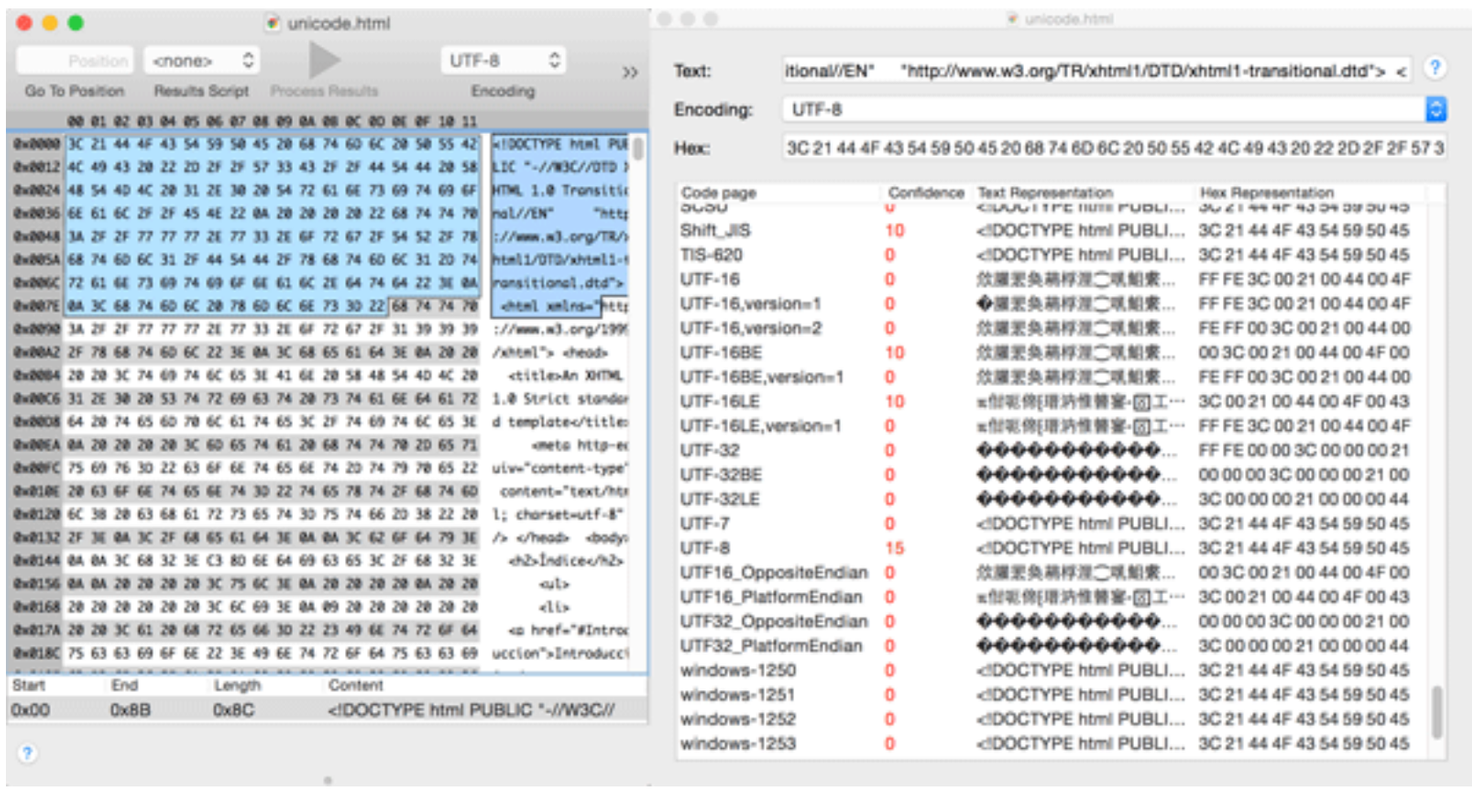
Deduce el formato a partir del punto de código de un caracter conocido. Por ejemplo, dado que un documento XML comienza con un caracter <, los puntos de código serán:

Encoding	Puntos de código
UTF-8	`U+3C`
UTF-16 Low Endian	`U+3C00`
UTF-16 Big Endian	`U+003C`
UTF-32 (BE)	`U+0000 U+003C`
UTF-32 (LE)	`U+3C00 U+0000`

Comprueba si la cadena o fichero comienza con un BOM. El BOM o **Marca de Orden de Byte** es un punto de código opcional que aparece al comienzo de un fichero Unicode. Indica el formato Unicode y el orden de sus bytes (endianness). En la práctica, la codificación más usada es UTF-8, y debido a que usa unidades de código de un byte, no necesita BOM. El estandar Unicode no requiere ni recomienda el uso de BOMs, pero define los siguientes:

Encoding	BOM (hexadecimal)
UTF-8	`EF BB BF`
UTF-16 Low Endian	`EF BB`
UTF-16 Big Endian	`FE FF`
UTF-32 (BE)	`00 00 FE FF`
UTF-32 (LE)	`FF FE 00 00`

Carga el texto en Synalize It! ($39), abre el comparador de encodings dandole a *Windows > Code Page Compare*, y pega parte del texto del documento en el campo *Text*. Te mostrará el texto interpretado en multitud de encodings, y podrás ver visualmente en cual se corresponde con lo que esperabas.



Editores hexadecimales

Cuando no entiendas lo que está pasando, quizas quieras ver el fichero en hexadecimal sin interpretación alguna.

hexdump es una herramienta BSD para mostrar un fichero en hexadecimal. Aquí muestro un fichero que contiene un BOM UTF-8 y la letra A.

```
~ $ hexdump A.txt
0000000 ef bb bf 41
0000004
```

Synalize It! es un editor hexadecimal que reconoce varias codificaciones de caracteres, y muestra como se ve un texto en multitud de codificaciones diferentes.

vi, tiene un modo de edición hexadecimal en el que puedes entrar y salir con:

```
:% ! xxd
:% ! xxd -r
```

Lenguajes de programación

ANSI y ISO-8859-1 siguen presentes en software creado cuando no existía Unicode o librerias maduras para procesarlo. Más raramente, también se usan porque es más rápido de procesar, o lo bastante bueno para comunicar información básica en software que no está de cara al usuario.

UTF-8 es con gran diferencia la codificación más usada. Los caracteres que coinciden con ISO-8859-1 usan un byte por caracter, el resto dos bytes, y casi siempre que encontremos más de dos, se tratará de un caracter emoji, o de caracteres japoneses, chinos, o koreanos.

Al programar delegaremos las complicaciones de Unicode a las librerías del sistema, pero debemos tener presente estos casos:

- Distinguir entre número de caracteres y número de bytes.
- Contar caracteres en cadenas que soportan caracteres combinados, pares sustitutos, y secuencias de variación.
- Comparar cadenas normalizadas para dar como iguales formas variantes de un mismo caracter.
- Nivel de soporte de Unicode en el motor de expresiones regulares. Hay tres niveles.
- Twitter por ejemplo, cuenta caracteres independientemente de los bytes necesarios para representarlos.

Objective-C

Los puntos de código se representan así:

- `@“\u266A"` (♪) para puntos de código del BMP (es decir, hasta `U+FFFF`).
- `@"\U0001F340"` (🍀) para el resto de puntos de código.

Las *cadenas* se representan con el objeto **NSString**. Los métodos de su API utilizan unidades de código UTF-16 (16 bits), con ordenación de bits (endianness) dependiente de la plataforma. Esto significa que su API se comporta como si su implementación interna fuese UTF-16, pero en realidad no está documentado internamente, y es posible que utilice la implementación que resulte más eficiente para cada texto.

Contar puntos de código, bytes, y número de caracteres:

```
// count code points, bytes, and characters
NSString *smiley = @"ðŸ˜€";
NSUInteger codePoints = [smiley length];
NSUInteger bytes = [smiley lengthOfBytesUsingEncoding:NSUTF8StringEncoding];
__block NSUInteger numberOfCharacters = 0;
[smiley enumerateSubstringsInRange:NSMakeRange(0, [smiley length])
                           options:NSStringEnumerationByComposedCharacterSequences
                        usingBlock:^(NSString *s, NSRange r, NSRange er, BOOL *stop) {
    numberOfCharacters++;
}];
NSLog(@"%lu, %lu, %lu", codePoints, bytes, numberOfCharacters); // 2, 4, 1
```

Comparar cadenas:

```
// compare variant forms with isEqual
NSString *a = @"\u00E9";  // Ã©
NSString *b = @"e\u0301"; // e + Â´
BOOL isEqual = [a isEqualToString:b]; // FALSE because it compares bytes
NSString *aCanonical = [a precomposedStringWithCanonicalMapping];
NSString *bCanonical = [b precomposedStringWithCanonicalMapping];
BOOL isEqualCanonical = [aCanonical isEqualToString:bCanonical]; // TRUE

// compare variant forms with localizedCompare:
NSString *c = @"ff"; // ff
NSString *d = @"\uFB00"; // ï¬€ ligature
NSComparisonResult result1 = [c localizedCompare:d];
NSComparisonResult result2 = [c compare:d options:NSLiteralSearch];
NSLog(@"%@, %@",
      result1==NSOrderedSame?@"same":@"different",  // same
      result2==NSOrderedSame?@"same":@"different"); // different
```

Buscar cadenas

```
rangeOfComposedCharacterSequencesForRange:
rangeOfComposedCharacterSequenceAtIndex:
```

Otros lenguajes de programación

Me queda pendiente hablar más de Unicode en Objective-C, Swift, Java, expresiones regulares, y compararlo con otros lenguajes de programación. Pero eso será otro día, porque hay un límite en la cantidad de Unicode que puede aguantar un

ser humano, y ya yo lo he alcanzado y superado.

HALA!, CADA UNO A SU CASA!

Referencias

- Unicode 7.0 Chapter One: Introduction
- UnicodeChecker Una herramienta para explorar y convertir Unicode.
- NSString y Unicode (objc.io)
- UTF-8 (Wikipedia)
- UTF-8, UTF-16, UTF-32 y BOM (unicode.org)
- Usted también puede hablar Unicode - Ross Carter
- Glosario de terminos Unicode (unicode.org)