Comencemos a programar con

VBA - Access

Apéndice 01

Tipos numéricos, Bytes y bits

Eduardo Olaz

Byte, bit, Bytes, bits ¿Qué es eso?

Los datos e instrucciones que utiliza el procesador los lee y escribe en la memoria.

La memoria está compuesta por unas celdas que pueden tomar dos estados **Activado** y **Desactivado** y que el procesador con la ayuda del sistema operativo puede leer y cambiar.

Estas celdas se agrupan de 8 en 8 unidades.

Cada celda representa a la unidad mínima de información, lo que equivale a un **bit**.

La agrupación de 8 celdas recibe el nombre de **Byte**.

La agrupación de varios Bytes recibe el nombre de **Palabra**.

Los procesadores Intel, y clónicos del tipo Pentium que se están a fecha de hoy utilizando en los PCs de esta generación, utilizan palabras de **4 bytes**, o lo que es lo mismo, trabajan mediante conjuntos de **32 bits**. Un procesador Pentium procesa bloques de información de 32 bits, y para ellos existen los sistemas operativos de 32 bits, como el Windows XP ó el Linux.

Los procesadores del los PCs de series anteriores, como los Intel 286 trabajaban con palabras de 2 Bytes, es decir con **16 bits**. Para ellos eran adecuados sistemas operativos como el Ms Dos.

Incluso los primeros Windows trabajaban con **16 bits**.

Ahora se están desarrollando microprocesadores de 64 bits, que son capaces de manejar palabras de 8 bytes, lo que los hace mucho más rápidos (y caros).

Los primeros PCs domésticos, entre los que podemos encontrar los del inefable **Alan Sugar** como los ZX 80, ZX 81, ZX Spectrum, Spectrum 128, SinclairQL, los de otros fabricantes con nombres tan singulares como Commodore, Atari, ACT Apricot, Dragon, Osborne, Júpiter, Acorn Atom, Tandy Radio Shack, Apple, Mattel Aquarius, la serie de ordenadores MMX, así como las primeras consolas de videojuegos, utilizaban procesadores de 8 bits que trabajaban con sistemas operativos adecuados, como el CPM.

Cuando todo esto ocurría, Años 80, muchos de vosotros justo acababais de nacer, y posiblemente a algunos de vosotros os quedarían aún años para venir al mundo.

Algunos de aquellos primeros ordenadores empezaron teniendo una memoria de 8 KBytes, lo que equivale a 8.192 Bytes, ó 65.536 celdillas de 1 bit.

Un siguiente paso que adoptaron estos ordenadores fue la memoria de 64 KBytes.

Estas cifras resultan ridículas cuando hablamos de las memorias de 1 Giga Byte que ya empiezan a ser consideradas como estándar en los PCs actuales.

Las limitaciones de memoria de esos equipos hacía que los programadores realizaran verdaderos trabajos de “orfebrería fina” con los programas, programas que vistos en la distancia nos resulta increíble que funcionaran con tan poca memoria. Como dato ilustrativo he tenido que ampliar la memoria de mi PC porque los 256 Mega Bytes que tenía hacían que diera problemas de “Falta de memoria”.

Kas, Megas, Gigas

Palabras como Kas, Megas, Gigas se han integrado en el lenguaje coloquial.

Pero ¿qué son realmente?.

En informática 1 KByte son 1.024 Bytes ó 2 elevado a la 10 Bytes

1 Mega ó Mega Byte son 1.024 K Bytes ó 1.048.576 Bytes ó 2 elevado a la 20 Bytes

1 Giga ó Giga Byte son 1.024 Mega Bytes ó 1.073.741.824 Bytes ó 2 elevado a la 30 Bytes

Hay otras medidas como Tera =1.024 gigas, etc, pero ahora no vienen al caso aunque, tal y como está avanzando todo esto, en fechas muy próximas seguro que se habla de discos con “Tantos Teras”.

Cómo afecta esto a la memoria del PC

Todos los datos, programas, imágenes, sonidos, etc… que maneja el PC pasan por la memoria como secuencias de unos y ceros.

La cantidad de bites simultáneos que sea capaz de manejar el procesador, junto con otros temas más complejos como paginaciones, etc… definirán la cantidad de memoria que es capaz de direccionar el procesador, y por tanto su capacidad máxima teórica de memoria.

Por direccionar entiéndase la cantidad de memoria que el procesador es capaz de leer y escribir.

No tiene nada que ver la memoria (64 kBytes con direccionamiento de 2 Bytes) manejable por un Z80 de 8 bits con la memoria manejable por un Pentium de 32 Bits.

Lectura / Escritura de los datos en memoria (Tipos numéricos)

Para dar una idea general de cómo maneja los datos la memoria voy a tratar de explicarlo de una forma simple.

Empecemos analizando cómo estamos acostumbrados a manejar los número en el sistema decimal.

Supongamos la cifra de este año:

El 2005

Si descomponemos la cifra 2005 nos da

2 unidades de mil más 5 unidades.

Es decir 2005 = 2 * 1000 + 5 * 1

También podríamos decir que 2005 = 2 * 1000 + 0 * 100 + 0 * 10 + 5 * 1

Si lo representáramos mediante potencias de 10 tendríamos.

2005 = 2 * 10^3 + 0 * 10^2 + 0 * 10^1 + 5 * 10^0

Nota:

Ya indicamos que el operador producto en programación suele ser el Asterisco [*]

El operador Exponente se representa mediante ^ . En otros lenguajes también se usa **

Haced ahora una prueba.

Abrid la calculadora de Windows.

En el menú Ver de ésta, seleccionad Científica.

Escribid ahora la cifra 2005.

A la izquierda, debajo de la ventana de las cifras podéis ver unos Botones de Opción, en los que pone Hex Dec Oct y Bin.

Por defecto está seleccionado **Dec**.

Esto indica que por defecto funciona con cifras en el formato decimal (de base 10).

Después de escribir 2005, seleccionad el Botón de opción **Bin**.

La cifra cambia a **11111010101**

Esta es la representación del valor 2005 en formato binario.

En realidad su representación es:

0000011111010101

Vamos a analizar esta cifra como hemos hecho con 2005 en decimal.

Así como **2 * 10^3 + 0 * 10^2 + 0 * 10^1 + 5 * 10^0**

Tenemos que

11111010101 =

1*2^10 + 1*2^9 + 1*2^8 + 1*2^7 + 1*2^6 + 0*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0=

2^10 + 2^9 + 2^8 + 2^7 + 2^6 + 2^4 + 2^2 + 2^0 =

1024 + 512 + 256 + 128 + 64 + 16 + 4 + 1 = **2005**

Si no os lo creéis haced la suma.

La diferencia entre el sistema binario que usan los ordenadores y el sistema decimal, es que el decimal usa potencias de 10, con coeficientes del 0 al 9, y el binario usa potencias de 2, usando como coeficientes el 0 y el 1.

El sistema hexadecimal usa potencias de 16 y sus cifras van del 0 al 9, más las letras A B C D E F para completar las cifras coeficiente desde el 10 al 15.

A vale 10, B vale 11, C 12, D 13, E 14, F 15.

Si después de escribir 2005 en la calculadora, en el modo decimal, seleccionáis la opción Hex en los Botones de Opción, veréis que cambia por **7D5**.

El sistema Hexadecimal trabaja en base 16

7D5 = 7 * 16^2 + 13 * 16^1 + 5 * 16^0 = 7 * 256 + 13 * 16 + 5 * 1 = 1792 + 208 + 5 = **2005**

Si ahora seleccionáis la opción Oct en los Botones de Opción, veréis que cambia a **3725**.

El sistema Octal trabaja en base 8.

3725 = 3 * 8^3 + 7 * 8^2 + 2 * 8^1 + 5 * 8^0 =

3 * 512 + 7 * 64 + 2 * 8 + 5 * 1 =

1536 + 448 + 16 + 5 = **2005**

Con todo esto ya tenemos las bases para entender cómo maneja la memoria ciertos datos.

Vamos a empezar con el tipo más básico, el tipo **Byte**.

El tipo **Byte** es un dato que puede ser almacenado en 8 bits, o como su nombre indica en un **Byte**.

Sus valores van del 00000000 → 0

al

11111111 → 255

Con esta estructura de bits podríamos, como ya he dicho, representar números del 0 al 255, e incluso si asignáramos a cada valor numérico una letra, representar hasta 255 caracteres.

El sistema de caracteres ASCII asigna caracteres a valores que están dentro de este rango.

Por ejemplo el carácter “a” tiene el valor 97. el “b” el valor 98, el “A” el 65, el “B” el 66, la cifra “0” el valor 48.

En entregas anteriores he hablado del valor Nulo ó Null. Pues bien, el Nulo se suele representar en memoria como un carácter de códico cero.

Si ahora tomáramos 2 bytes y utilizáramos sus 16 bits tendríamos posibilidad de representar del 0000000000000000 → 0

al

1111111111111111 → 65535

Si asignamos a valores dentro de este rango, caracteres de texto concretos podríamos representar prácticamente todos los caracteres que existen en los diferentes idiomas, incluyendo el Chino y el japonés.

Algo así es lo que hace el sistema de caracteres UNICODE.

Igualmente podríamos almacenar valores enteros del 0 al 65535. Este tipo de dato numérico no existe en VBA, pero sí por ejemplo en C# ó VB.Net y recibe el nombre de **Char**.

En VBA existe el dato llamado **Integer**, que utiliza para el rango numérico los 15 bits de la derecha, con lo que se pueden representar valores que van del

000000000000000 → 0

al

111111111111111 → 32767

El primer bit se utiliza para el signo. Luego hace una serie de operaciones a nivel de bits, pero para nosotros eso no es transcendente.

En realidad el tipo **Integer** puede tomar valores en el rango que va del **- 32.768** al **32.767** usando para ello **2 Bytes**.

De forma semejante tenemos otro número que maneja enteros, pero de más precisión.

Es el llamado **Long**.

El tipo **Long** usa 31 bits para el valor numérico más el primer bit para el signo

Con 31 bits podríamos representar valores del

00000000000000000000000000000000 → 0

al

11111111111111111111111111111111 → 2147483647

Con el bit del signo, el tipo **Long** puede tomar valores en el rango que va del **- 2147483648** al **2147483647** usando para ello **4 Bytes**.

Hay un caso muy especial de dato, y es el llamado **Currency** (Tipo moneda).

El tipo **Long** usa 63 bits para el valor numérico más el primer bit para el signo.

Con 63 bits se pueden representar valores del 0 al 9223372036854775803.

Usando un bit adicional para el signo, el tipo currency es capaz de representar en realidad datos que van del -922.337.203.685.477,5808 al 922.337.203.685.477,5807.

El tipo Currency tiene la peculiaridad de que maneja 4 cifras decimales.

En realidad no es un tipo de los llamados de Coma flotante, sino que se almacena internamente como si fuera un valor numérico de cifras enteras, y luego en los cálculos considera que las 4 cifras decimales menos significativas están a la derecha de la coma.

¿Cómo podríamos representar un valor no entero, por ejemplo el 0,25?

Supongamos que quisiéramos representar valores menores que 1 y mayores que 0 usando para ello 1 Byte.

Podríamos hacer lo siguiente

El primer bit representaría al valor 1/2, el segundo a 1/4, el tercero 1/8, el cuarto 1/16, y así hasta el 1/256

Para representar el valor 0,25 nos bastaría con el bit 2º, con lo que el valor 0,25, mediante nuestro método se representaría así: 01000000

Si quisiéramos representar 0,75 nos bastaría con el primer bit más el segundo

Lo que equivaldría a 0,75 = 0,5 + 0,25 Es decir 11000000

Para algunas cifras tendríamos un problema de precisión. Por ejemplo ¿cómo representar el valor 0,3?. En el sistema decimal es muy sencillo y se hace 3 * 0,1 es decir 0,3.

En cambio con 8 bits la forma más aproximada sería:

0 + 1/4 + 0 + 0 + 1/32 + 1/64 + 0 + 1/256 → 0,30078125 lo que equivaldría a 01001101

La combinación, sin pasarse de 0,3 más aproximada sería

0 + 1/4 + 0 + 0 + 1/32 + 1/64 + 0 + 0 → 0,296875 lo que equivaldría a 01001100

Si quisiéramos representar el número 23,3 podríamos usar 16 bits (2 Bytes) el primero para la parte entera, y el segundo para la parte decimal.

Con este sistema quedaría 23,3 → 00010111 01001100

Este procedimiento, que yo sepa, no lo utiliza ningún ordenador. En la vida real se usan métodos más sofisticados, pero la base de su funcionamiento es esta. Ver

Como se puede comprender, la precisión y la magnitud que se pueden representar es directamente proporcional al número de Bytes utilizados.

Así por ejemplo una variable declarada como de tipo **Single** utiliza 4 Bytes. Esto le permite manejar números en los rangos que van de -3,402823E38 a -1,401298E-45 para valores negativos y de 1,401298E-45 a 3,402823E38 para valores positivos

Una variable de tipo **Double**, utiliza 8 Bytes. Sus datos se almacenan siguiendo el formato IEEE 754.

El siguiente texto está sacado del sitio correspondiente a Tutorías de la UNED
http://www.etsimo.uniovi.es/~antonio/uned/ieee754/formato.html

En él se explica la estructura del formato IEEE 754.

Es una información para los que quieren de verdad conocer “las tripas de la cosa”, aunque reconozco que su digestión no es precisamente sencilla. Como aclaración el operador **, es el operador Exponente. Así la expresión 2**(129-127) indica que se eleva 2 al cuadrado.

Lo indicado como Simple precisión sería aplicable al tipo **Single**, y por doble precisión al tipo **Double**.

No os preocupéis si no entendéis lo aquí expuesto; lo pongo como “información avanzada”.

---

*El **IEEE** (Instituto de Ingenieros Eléctricos y Electrónicos) ha creado un estándar la presentaciónd de números en coma flotante. Este estándar especifica como deben representarse los números en coma flotante con simple precisión (32 bits) o doble precisión (64 bits), y también cómo deben realizarse las operaciones aritméticas con ellos.*

Simple Precisión

*El estándar **IEEE-754** para la representación en simple precisión de números en coma flotante exige una cadena de 32 bits. El primer bit es el bit de signo (S), los siguientes 8 son los bits del exponente (E) y los restantes 23 son la mantisa (M):*

```
S EEEEEEEE MMMMMMMMMMMMMMMMMMMMMMM
0 1      8 9                     31
```

El valor V representado por esta cadena puede ser determinado como sigue:

- *Si E=255 y M es no nulo, entonces V=NaN ("Not a number")*
- *Si E=255 y M es cero y S es 1, entonces V=-Infinito*
- *Si E=255 y M es cero y S es 0, entonces V=Infinito*
- *Si 0<E<255 entonces V=(-1)**S * 2 ** (E-127) * (1.M)*
  donde "1.M" se emplea para representar el número binario creado por la anteposición a M de un 1 y un punto binario.
- *Si E=0 y M es no nulo, entonces V=(-1)**S * 2 ** (-126) * (0.M)*
  Estos son valores "sin normalizar".
- *Si E=0 y M es cero y S es 1, entonces V=-0*
- *Si E=0 y M es cero y S es 0, entonces V=0*

En particular,

```
  0 00000000 00000000000000000000000 = 0
  1 00000000 00000000000000000000000 = -0

  0 11111111 00000000000000000000000 = Infinito
  1 11111111 00000000000000000000000 = -Infinito

  0 11111111 00000100000000000000000 = NaN
  1 11111111 00100010001001010101010 = NaN

  0 10000000 00000000000000000000000 = +1 * 2**(128-127) * 1.0 = 2
  0 10000001 10100000000000000000000 = +1 * 2**(129-127) * 1.101 =
6.5
  1 10000001 10100000000000000000000 = -1 * 2**(129-127) * 1.101 = -
6.5

  0 00000001 00000000000000000000000 = +1 * 2**(1-127) * 1.0 = 2**(-
126)
  0 00000000 10000000000000000000000 = +1 * 2**(-126) * 0.1 = 2**(-
127)
  0 00000000 00000000000000000000001 = +1 * 2**(-126) *
                                       0.00000000000000000000001 =
                                       2**(-149)  (valor positivo más
pequeño)
```

Doble precisión

El estándar IEEE-754 para la representación en doble precisión de números en coma flotante exige una cadena de 64 bits. El primer bit es el bit de signo (S), los siguientes 11 son los bits del exponente (E) y los restantes 52 son la mantisa (M):

```
S EEEEEEEEEEE MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
0 1          11 12                                                63
```

El valor V representado por esta cadena puede ser determinado como sigue:

- *Si E=2047 y M es no nulo, entonces V=NaN ("Not a number")*
- *Si E=2047 y M es cero y S es 1, entonces V=-Infinito*
- *Si E=2047 y M es cero y S es 0, entonces V=Infinito*
- *Si 0<E<2047 entonces V=(-1)**S * 2 ** (E-1023) * (1.M)*
  donde "1.M" se emplea para representar el número binario creado por la anteposición a M de un 1 y un punto binario.
- *Si E=0 y M es no nulo, entonces V=(-1)**S * 2 ** (-1022) * (0.M)*
  Estos son valores "sin normalizar".
- *Si E=0 y M es cero y S es 1, entonces V=-0*
- *Si E=0 y M es cero y S es 0, entonces V=0*

Bibliografía:

ANSI/IEEE Standard 754-1985,
Standard for Binary Floating Point Arithmetic

Tipo Decimal

Dentro de los tipos numéricos hay un tipo muy especial. Es el tipo **Decimal**.

Este tipo necesita de 12 Bytes para su representación, o lo que es lo mismo 96 bits.

Tal cantidad de bits le permite manejar números de un tamaño y precisión extrema.

El tipo Decimal permite definir el número de decimales que va a utilizar.

El número de decimales se puede establecer entre 0 y 28.

Según la ayuda los rangos de valores máximos y mínimos que puede manejar son

+/-7,9228162514264337593543950335 y +/-0,0000000000000000000000000001

El inconveniente de este tipo de número es que para manejarlo precisa declarar primero la variable como **Variant** y luego hacer una conversión a Decimal mediante la función de Conversión de Tipo **CDec**.

Para probarlo vamos a crear un procedimiento Sub en un módulo estandar

```
Public Sub PruebaDecimal()
    Dim varDecimal As Variant
    varDecimal = CDec(varDecimal)
    varDecimal = CDec(1) / 3
    Debug.Print varDecimal
End Sub
```

Si lo llamamos desde la ventana Inmediato, escribiendo en ella `PruebaDecimal` y presionando la tecla [Intro], nos imprimirá:

```
0,6666666666666666666666666667
```

Es decir un resultado con una precisión de 28 decimales.

¿Por qué he hecho la conversión a Decimal de la cifra 1? en la expresión:

```
varDecimal = CDec(2) / 3
```

Porque para que a la variable **`varDecimal`** se le pueda asignar un valor decimal, la cantidad a asignar debe ser Decimal. Esto implica que en la operación asignada, al menos uno de los operandos debe ser convertido a Decimal.

Si esa línea de código la hubiéramos escrito así

```
varDecimal = 2 / 3
```

Nos hubiera impreso sólo

```
0,666666666666667
```

Que está en el rango del **Double**, no del **Decimal**.

Por ejemplo si quisiéramos asignar a varDecimal el resultado de la operación

`2/3 + 4` deberíamos hacer, por ejemplo

```
varDecimal = CDec(2) / 3 + 4
```

En cambio si hacemos

```
varDecimal = 2 / 3 + CDec(4)
```

Nos devolvería un dato de tipo Double.

Por ello, para no tener errores, si es que necesitáramos utilizar variables Decimales, por ejemplo si nos contratara la Nasa para programar la trayectoria de un viaje estelar, o tuviéramos que efectuar cálculos estadísticos muy complejos, convendría hacer la conversión de todas las operaciones al formato Decimal antes de asignarlas a una variable.

Por suerte hay pocos casos en los que serían necesarios este tipo de valores.

Manejo de textos

Los textos se almacenan carácter a carácter

Windows maneja unas tablas de caracteres que asignan a cada carácter un número del tipo Byte. (entre 0 y 255)

Los 128 primeros códigos de Windows guardan correlación con los 128 que utilizaba MsDos.

Por ejemplo, si a la variable `strNombreApellido` declarada como de tipo **String**, le asignamos mis datos:

```
strNombreApellido = "Eduardo Olaz"
```

En la memoria quedará almacenado de la siguiente manera

```
 69 100 117  97 114 100 111  32  79 108  97 122
 E   d   u   a   r   d   o       O   l   a   z
```

(Al espacio en blanco se le asigna el valor 32).

Es el propio Windows el que define los códigos a aplicar.

Hay discrepancias entre algunos de los caracteres superiores al 127, es el caso de las eñes y las vocales acentuadas. No obstante se pueden utilizar funciones de conversión definidas en el API de Windows, como son:

`OemToChar` cuya cabecera es

```
Declare Function OemToChar _
            Lib "user32" _
            Alias "OemToCharA" ( _
            ByVal CadenaAConvertir As String, _
            ByVal CadenaConvertida As String) _
            As Long
```

Y **CharToOem** de cabecera:

```
Declare Function CharToOem _
            Lib "user32" _
            Alias "CharToOemA" ( _
            ByVal CadenaAConvertir As String, _
            ByVal CadenaConvertida As String) _
            As Long
```

Si tuviéramos que realizar cambios entre “textos de Dos” como serían los que almacena DBase y textos Windows, o a la inversa para imprimir de forma directa en impresoras de agujas o grabar en ficheros DBase podríamos declarar unas funciones de conversión.

El código para estas operaciones podría ser el siguiente:

```
Declare Function CharToOem _
            Lib "user32" _
            Alias "CharToOemA" ( _
            ByVal CadenaAConvertir As String, _
            ByVal CadenaConvertida As String) _
            As Long


Declare Function OemToChar _
            Lib "user32" _
            Alias "OemToCharA" ( _
            ByVal CadenaAConvertir As String, _
            ByVal CadenaConvertida As String) _
            As Long

Public Function TextoWindowsADos( _
            ByVal Cadena As String) _
            As String
    Dim strBuffer As String
    Dim Resultado As Long
    strBuffer = String(Len(Cadena), " ")
    Resultado = CharToOem(Cadena, strBuffer)
    TextoWindowsADos = strBuffer
End Function

Public Function TextoDosAWindows( _
            ByVal Cadena As String) _
            As String
    Dim strBuffer As String
    Dim Resultado As Long
    strBuffer = String(Len(Cadena), " ")
    Resultado = OemToChar(Cadena, strBuffer)
    TextoDosAWindows = strBuffer
End Function
```