# Basic Express
# Referencia del Lenguaje

**Versión 2.0**





Traducción española: Alicia Bernal, Revisión: Pablo Pompa
www.superrobotica.com

2.00.H

# Contenido

# 1. Información general

## 1.1 Módulos

*Los módulos* le permiten dividir un programa en múltiples ficheros. Al utilizar módulos, puede controlar la visibilidad de los subprogramas, que pueden ser públicos (globales) o privados para un módulo. Las variables y las constantes pueden también ser globales, locales para los módulos, o locales para los subprogramas.

Un subprograma público puede ser llamado desde otros módulos. Un subprograma privado sólo puede ser llamado desde dentro del módulo en el que aparece.

Las variables públicas y privadas que aparezcan en el código de nivel de módulo (*module level)* tienen propiedades similares, al igual que las constantes públicas y privadas. El código de nivel de módulo se refiere al código que no pertenece a ningún subprograma, y que aparece al principio de un módulo.

Ejemplo de un módulo:

```
Public A As Integer
Privad B As Single ' El código de nivel de módulo termina aquí.

Public Sub Main()

   Dim K As Integer

   A = 1
   For K = 1 a 10
      A = A + 1
   Next
   B = CSng(A)
   Call Square(B)

End Sub

Privad Sub Square(X As Single)

   X = X * X

End Sub
```

En este ejemplo, la variable A y el procedimiento Main son visibles en todos los otros módulos. La variable B y el procedimiento Square son sólo visibles desde dentro de este módulo. La variable K es una variable local visible sólo desde el procedimiento Main.

Las variables del código del nivel de módulo también retienen sus valores entre las llamadas de subprogramas. Estas variables reciben algunas veces el nombre de variables *estáticas*. Por el contrario, las variables locales como K no retienen sus valores entre las llamadas.

Los nombres de los módulos se toman de los nombres de los ficheros, lo que significa que los nombres de los ficheros (menos las extensiones) deben ser identificadores legales de Basic.

Identificadores: Todos los identificadores de BasicX deben comenzar por una letra, y el resto de los caracteres deben ser también letras, dígitos o guiones bajos. Los identificadores son sensibles al uso de mayúsculas o minúsculas. Un identificador puede tener hasta 255 caracteres de longitud, y todos ellas serán relevantes.

## 1.2 Programa principal (Main program)

El programa activa la ejecución de un procedimiento denominado Main, que debe ser un procedimiento público.

Puede tener más de un procedimiento *privado* llamado Main, siempre que no disponga de más de un Main por módulo. Sin embargo, es necesario un único procedimiento *público* Main—de lo contrario el programa no sabrá dónde iniciarse.

## 1.3 Formato de comandos

Para hacer que un comando largo sea más fácil de leer, éste puede ocupar dos o más líneas añadiendo un carácter de continuación de línea (guión bajo) como el último carácter de la línea.

El guión bajo debe estar precedido por uno o más espacios en blanco o etiquetas, y debe ser el último carácter en una línea. Ningún elemento, ni siquiera los comentarios, puede seguir en la misma línea. Los comentarios en concreto no pueden ocupar más de una línea de esta manera.

Ejemplo:

```
A = A + B + Eval( _
   C, D, E, F, _
   G, H, I)
```

Los literales de cadena tampoco pueden ocupar más de una línea de la manera indicada, aunque pueden utilizar operadores de concatenación para dividir una cadena en la siguiente línea. En el siguiente ejemplo, ambas líneas son equivalentes:

```
S = "Hello, world"

S = "Hello, " & _
   "world"
```

## 1.4 Formato de comentario

Se utiliza un apóstrofe para marcar los comentarios.
Ejemplo:

```
I = 32767 ' Comentario
```

Se ignorarán todos los caracteres situados a la derecha del apóstrofe.

# 2. Subprogramas

## 2.1 Información general

Un subprograma le permite tomar un grupo de comandos relacionados y tratarlos como una única unidad.

Los subprogramas están formados por procedimientos y funciones. La diferencia entre un procedimiento y una función es que una función puede aparecer como parte de una expresión, aunque un procedimiento debe llamarse en un comando independiente.

No hay un límite específico para el nivel anidado de llamadas de un subprograma – el único límite es la capacidad disponible de la memoria RAM. Las llamadas de los subprogramas pueden también ser recursivas.

## 2.2 Procedimientos secundarios.

```
[Privad|Public] Sub procedure_name (arguments)
   [statements]
End Sub
```

Puede llamar a un procedimiento utilizando la palabra clave *Call*. Puede salir de un procedimiento utilizando un comando Exit Sub.

Ejemplo:

```
Privad Sub GetPosition(ByRef X As Single)

   Call ReadDevice(X)
   If (X > 100.0) Then
      Exit Sub
   End If
   X = X * X

End Sub
```

Cuando llama a un procedimiento, la palabra clave *Call* es opcional. Si se omite Call, los paréntesis alrededor de los parámetros reales (si los hubiera) deben omitirse también. Por ejemplo, los dos comandos siguientes son equivalentes:

```
Call GetPosition(X)
GetPosition X
```

## 2.3 Funciones

```
[Privad|Public] Function function_name (arguments) As type
   [statements]
End Function
```

Una función es un subprograma similar a un procedimiento. La diferencia reside en que se llama a una función utilizando su nombre dentro de una expresión, seguido de la lista de argumentos de la función (si la hubiera) entre paréntesis.

La función devuelve un valor, que puede estar definido al asignar el nombre de la función dentro de la función propiamente dicha. En otras palabras, puede tratar el nombre como si se tratara de una variable local, y normalmente se puede leer y escribir el nombre desde dentro de la función. Una excepción son las funciones de cadena, en las que la función devuelve un valor de sólo escritura (write-only) dentro de la función.Ejemplo:

```
Public Función F(ByVal i As Integer) As Integer

   F = 2 * i   ' Esto define el valor que devuelve la función.

   F = F * F   ' Puede también leer el nombre de la función.

End Función
```

Puede también salir de una función utilizando un comando Exit Function.
Ejemplo:

```
Function F(ByVal i As Integer) As Single

   If (i = 3) Then
      F = 92.0
      Exit Function
   End If

   F = CSng(i) + 1.0

End Function
```

## 2.4 Parámetros de conexión con subprogramas

Las funciones pueden devolver tipos no persistentes escalares o tipos de cadenas.

Ejemplo de sintaxis de funciones de cadena:

```
Function F() As String
   F = "Hello, world"  ' F es sólo de escritura.
End Function
```

Para obtener una mayor eficiencia una función de cadena devuelve un valor de sólo escritura (write-only). Es decir, se puede asignar la devolución de la función, pero no se puede leer, ni utilizarla en una expresión o pasarla a otro subprograma. Cada asignación de la devolución de la función debe estar seguida inmediatamente por un comando "Exit Function" o "End Función".

Si una función devuelve un objeto de número entero sin signo (UnsignedInteger) o un objeto largo sin signo (UnsignedLong), el primer comando de la función debe ser un comando tipo *Set*.

Ejemplo:

```
Function F() As UnsignedInteger
   Set F = New UnsignedInteger
```

```
    [...]
End Function
```

**¿Cómo se pasan parámetros a subprogramas?**

Es posible pasar parámetros a un subprograma a través de la referencia (ByRef) o del valor (ByVal).

**Paso por referencia** – si pasa un parámetro por referencia, cualquier cambio realizado al parámetro se volverá a transmitir a la persona que llama. El paso por referencia está definido por defecto.

**Paso por valor** – si pasa un parámetro por valor, no se permitirán que los cambios se transmitan a la persona que llama. Con algunas excepciones, si un argumento se pasa por valor, se hace una copia del argumento, y el subprograma llamado operará en la copia. Los cambios realizados a la copia no afectan a la llamada.

Una excepción son los parámetros de cadena pasados por valor. Para obtener una mayor eficiencia no se hace la copia de la cadena. En su lugar, la cadena está protegida contra escritura en el subprograma llamado, lo que implica que no se puede ni asignar la cadena ni pasarla a otro subprograma. Sin embargo, está permitido pasarla por valor a otro subprograma.

La otra excepción es para los tipos de variables de entero sin signo (UnsignedInteger) y largo sin signo (UnsignedLong), que son tratados de la misma manera – estos parámetros están protegidos contra escritura en los subprogramas llamados.

**Parámetros reales y parámetros formales** –el tipo y el número de los parámetros reales deben coincidir con el tipo y el número de los parámetros “formales” (éstos aparecerán en la declaración del subprograma). Si se produce alguna falta de coincidencia, el compilador advertirá un error. No realizará conversiones implícitas de los tipos.

Ejemplo de sintaxis:

```
Sub Collate(ByVal I As Integer, ByRef X As Single, ByVal B As Byte)
```

En el ejemplo siguiente, SortList es una matriz:

```
Function MaxValue(ByVal SortList() As Byte, S As String) As Integer
```

Observe que la cadena S se ha pasado por referencia, que es la opción por defecto.

**Restricciones de los mecanismos de pasos:**

1. Los elementos de las variables escalares y de las matrices pueden pasarse tanto por valor como por referencia.

2. Una matriz puede pasar únicamente por referencia. La matriz debe ser también de una sola dimensión y tener un límite inferior de uno. Es imposible pasar cualquier otro tipo de matriz, sea cual sea el mecanismo de paso.

3. Las expresiones numéricas y los literales numéricos pueden pasarse por valor pero no por referencia. Lo mismo es aplicable a las expresiones y literales booleanos.

4. Las variables persistentes pueden pasar únicamente por valor, y sólo después de que hayan sido convertidos al tipo no persistente* correspondiente.

5. Si utiliza el paso por valor para los tipos String, UnsignedInteger y UnsignedLong, los parámetros están protegidos contra escritura dentro de los subprogramas llamados, lo que implica que no puede cambiar sus valores ni utilizarlos como contadores de bucle For-Next. Si pasa estos parámetros a otro subprograma, deberá ser por valor y no por referencia.

6. No está permitido que las funciones acepten expresiones generales de cadenas, incluyendo a los literales de cadena, como parámetros reales. Si pasa una cadena a una función, la cadena debe ser una variable, incluso si se pasa por valor. (Esta restricción no se aplica a los procedimientos.)

- Aunque generalmente BasicX no realiza de manera implícita conversiones de tipos, convertirá automáticamente una variable persistente al tipo de variable no persistente que corresponda cuando la variable pase como el argumento de un subprograma. Por ejemplo, una variable PersistentInteger se convierte a Integer.

**Resumen:**

| Parámetro | ByRef | ByVal |
|---|---|---|
| Variable escalar | Sí | Sí |
| Elemento de matriz | Sí | Sí |
| 1D matriz, límite inferior = 1 | Sí | No |
| Matriz multidimensional | No | No |
| Matriz con límite inferior que no sea 1 | No | No |
| Expresión numérica | No | Sí |
| Literal numérico | No | Sí |
| Expresión booleana | No | Sí |
| Literal booleano | No | Sí |
| Variable persistente | No | Sí |

# 3. Estructuras de control

## 3.1 Comandos tipo If-Then

```
If (boolean_expression) Then
   [statements]
End If

If (boolean_expression) Then
   [statements]
Else
   [statements]
End If

If (boolean_expression) Then
   [statements]
ElseIf (boolean_expression) Then
   [statements]
[ElseIf (boolean_expression) Then]
```

```
    [statements]
[Else]
    [statements]
End If
```

Ejemplos:

```
If (i = 3) Then
    j = 0
End If

If (i = 1) Then
    j = 3
ElseIf (i = 2) Then
    j = 4
Else
    j = 5
End If
```

## 3.2 Comandos Do-Loop

Bucle infinito:

```
Do
    [statements]
Loop
```

Puede añadir un cualificador "While", que comprueba una expresión booleana para determinar si se ejecuta o no el cuerpo del bucle. El bucle continúa mientras la expresión siga siendo de tipo verdadero (true). Se puede hacer un test al principio o al final del bucle. Si el test está al final, el cuerpo siempre se ejecuta al menos una vez.

El cualificador "Until" es similar a "While", excepto que la lógica es invertida. Es decir, el bucle se *termina* en lugar de continuar cuando la expresión booleana es de tipo verdadero (true).

Ejemplos:

```
Do While (boolean_expression)
    [statements]
Loop

Do
    [statements]
Loop While (boolean_expression)

Do Until (boolean_expression)
    [statements]
Loop

Do
    [statements]
Loop Until (boolean_expression)
```

El comando Exit Do puede utilizarse como un bucle Do-Loops. Es posible anidar los bucles Do-Loops hasta en 10 niveles.

Ejemplos:

```
Do
   j = j + 1
   If (j > 3) Then
      Exit Do
   End If
Loop

Do While (j < 10)
   j = j + 1
Loop

Do
   j = j + 1
Loop Until (j <= 10)
```

## 3.3 Comando For-Next

Un comando For-Next puede utilizarse para ejecutar un bucle un número determinado de veces.

```
For loop_counter = start_value To end_value [Step 1 | -1]
   [statements]
Next
```

El bucle se ejecuta desde el valor start_value hasta end_value en pasos de +1 ó -1. Si esto arroja un valor cero (zero) o valor negativo, el bucle no se ejecuta de ninguna manera.

Loop_counter debe ser un tipo discreto. Así mismo, loop_counter, start_value y end_value deben ser todos del mismo tipo. Tanto start_value como end_value pueden ser expresiones.

Los contadores de bucle deben ser variables locales, y no está permitido modificar el contador desde dentro del bucle. Es decir, un contador debe tratarse como si se tratase de una constante dentro del bucle. Si desea pasar el contador a un subprograma, debe pasarlo por valor (by value). No está permitido pasarlo por referencia (by reference) – de lo contrario el subprograma llamado podría modificar al contador de manera indirecta.

Al final de un bucle de tipo For-Next, el contador se supone que es no definido por defecto. De acuerdo con esto, el margen de acción de cada contador está limitado al bucle al que pertenece, con la excepción de los bucles múltiples no anidados, a los que se les permite compartir los mismos contadores.

En otras palabras, no es posible utilizar los contadores de los bucles fuera de sus bucles.

El comando "Exit For" puede utilizarse para salir de un bucle tipo For-Next.

Los bucles For-Next pueden anidarse en unos niveles de diez.

Ejemplos:

```
For i = 1 To 10
   j = j + 1
Next
```

Este bucle se ejecuta diez veces, con incrementos cada una de las veces que se ejecuta.

El siguiente ejemplo ilustra un comando de tipo Exit-For. El contador del bucle va descendiendo, y el número máximo de interacciones es diez.

```
For i = 10 To 1 Step -1
   j = j - 1
   If (j < 3) Then
      Exit For
   End If
Next
```

## 3.4 Comando Select-Case

Un comando Select-Case puede utilizarse para seleccionar una opción entre una lista de alternativas. Sintaxis:

```
Select Case test_expression
   Case expression_list1
      [statements]
   [Case expression_list2]
      [statements]
   [Case Else]
      [statements]
End Select
```

La expresión test_expression se evalúa una vez, en la parte superior del bucle. A continuación, el programa remite a la primera expresión expression_list de la lista coincida con el valor de test_expression, y se ejecuta el bloque de comandos asociados.

Si no coincide ningún valor, el programa remite al comando opcional Case Else, si está presente. En caso de que no haya ningún comando Case Else, el programa remite al comando que esté detrás del comando End Select.

La expresión test_expression debe ser discreta, y de tipo non-string (es decir, numérico booleano o discreto). Cada opción expression_list debe tener el mismo tipo que test_expression.

Ejemplo:

```
Select Case BinNumber(Count)
   Case 1
      Call UpdateBin(1)
   Case 2
      Call UpdateBin(2)
   Case 3, 4
      call EmptyBin(1)
      call EmptyBin(2)
   Case 5 To 7
      Call UpdateBin(7)
   Case Else
```

```
        Call UpdateBin(8)
    End Select
```

### 3.5 Comando GoTo

Un comando GoTo remite incondicionalmente a la etiqueta especificada.
Ejemplo:

```
    GoTo label_name

label_name:
```

Las etiquetas deben estar seguidas por dos puntos (:).

# 4. Variables, constantes y tipos de datos

## 4.1 Tipos de datos

| Tipo | Almacenamiento | Rango |
|---|---|---|
| Boolean (Booleano) | 8 bits | Verdadero ---Falso (True .. False) |
| Byte (Byte) | 8 bits | 0 .. 255 |
| Entero (Integer) | 16 bits | -32 768 .. 32 767 |
| Largo (Long) | 32 bits | -2 147 483 648 .. 2 147 483 647 |
| Single (Sencillo | 32 bits | -3.402 823 E+38 .. 3.402 823 E+38 |
| Cadena (String) | Depende | De 0 a 64 caracteres |
| BoundedString (Cadena limitada) | Depende | De 0 a 64 caracteres |

**Almacenamiento de cadenas** – hay tres tipos diferentes de cadenas – cadenas de longitud variable, cadenas de longitud fija, y cadenas limitadas. La longitud máxima permitida de cada una de las cadenas depende de la configuración del compilador. El rango seleccionable es de 1 a 64 caracteres, con un valor por defecto de 64.

El almacenamiento requerido para una cadena de longitud fija es de 2 más el número de caracteres especificados en la declaración, hasta el número de caracteres definido por el usuario. Una cadena de largo variable siempre requiere un almacenamiento constante, que es 2 bytes más el número máximo de caracteres definido por el usuario. Una cadena ilimitada requiere una longitud declarada más 2 bytes para el almacenamiento.

En este ejemplo, se supone que el compilador está configurado con un límite de 45 caracteres:

```
' Una cadena de longitud fija requiere 2 + 5 = 7 bytes de almacenamiento.
Dim A As String * 5

' Una cadena de longitud variable requiere 2 + 45 = 47 bytes de almacenamiento.
Dim B As String
```

### ***Cadenas limitadas*** *– la sintaxis es la siguiente:*

```
Dim B As BoundedString_N  ' Donde N es la longitud máxima.
```

Ejemplos:

```
Dim A As BoundedString_10  ' Permite hasta 10 caracteres.
Dim A As BoundedString_64  ' Permite hasta 64 caracteres.
```

## 4.2 Declaraciones

Todas las variables deben estar declaradas antes de utilizarlas. Las declaraciones implícitas no están permitidas.

Para variables declaradas en el código de nivel de módulo:

```
[Public | Privad | Dim] variable As type
```

Las variables públicas son globales y visibles a lo largo de todo el programa. Las variables privadas son visibles únicamente en el módulo en el que estas aparecen. Por defecto las variables son privadas.

Para las variables declaradas dentro de un subprograma:

```
Dim variable As type
```

Las variables declaradas dentro de un subprograma son visibles únicamente dentro del subprograma.

Ejemplos:

```
Public Distance As Integer    ' Variable de nivel de módulo, global
Privad Temperature As Single ' Variable de nivel de módulo, local para el módulo

Sub ReadPin()
   Dim PinNumber As Byte     ' La variable es local para este subprograma
End Sub
```

Las cadenas pueden tener una longitud fija o variable. Ejemplo de sintaxis:

```
Dim S1 As String      ' Longitud variable
Dim S2 As String * 1  ' Cadena de 1 carácter
Dim S2 As String * 64 ' Cadena de 64 caracteres
```

Una cadena de longitud variable puede contener de 0 a 64 caracteres, dependiendo de la configuración del compilador.

## 4.3 Constantes

Para las constantes declaradas en el código de nivel de módulo:

```
[Public | Privad] Const constant_name As type = literal
```

El valor por defecto es privado.

Para constantes declaradas dentro de un subprograma:

```
Const constant_name As type = literal
```

Ejemplos:

```
Const Pi As Single = 3.14159
Privad Const RoomTemperature As Single = 70.0
Public Const MaxSize As Byte = 20
Const SwitchOn As Boolean = True, SwitchOff As Boolean = False
```

## 4.4 Literales numéricos

Ejemplos de números enteros decimales:

```
1
-1
10
255
```

Tenga en cuenta que está permitido que los literales numéricos enteros tengan un punto decimal.

Ejemplos de puntos decimales flotantes:

```
1.0
-0.05
1.53E20
-978.3E-3
```

Ejemplos de enteros hexadecimales:

```
&H3
&HFF
&H7FFF     ' 32767
-&H8000&   ' -32768 (tenga en cuenta la presencia del signo &)
```

Los signos & son necesarios los signos & para los números hexadecimales dentro del rango &H8000 a &HFFFF (Desde 32 768 a 65 535).

Constantes binarias:

```
bx00000001 ' 1
bx00001111 ' 15
bx11111111 ' 255
```

Las constantes binarias tienen un prefijo "bx" seguido por un número binario de 8 dígitos. De una manera estricta, estas entidades no son literales numéricos, aunque se tratan como equivalentes para las constantes simbólicos del tipo Byte.

## 4.5 Conversión de tipos de datos

| Función | Resultado |
|---|---|
| CBool | Boolean |
| CByte | Byte |
| CInt | Integer |
| CLng | Long |
| CSng | Single |
| CStr | String |
| FixB | Byte |
| FixI | Integer |
| FixL | Long |

CBool sólo permite el tipo Byte como operador.

FixB, FixI y FixL sólo permiten tipos de puntos flotantes como operadores. Estas tres funciones utilizan la truncación para convertir a tipos discretos (ver también Fix, que es similar aunque devuelve un valor de punto flotante).

Las otras funciones utilizan cualquier tipo numérico como operadores, aunque con la siguiente diferencia – cuando se utiliza CByte, CInt y CLng con operadores de punto flotante, se utiliza el redondeo estadístico (statistical rounding). Esto implica que el número se convierte al número entero más próximo, a menos que dicho número se encuentre exactamente a medio camino entre dos números enteros, en cuyo caso se redondeará al número *par* más cercano. Ejemplos:

| F | CInt(F) |
|---|---|
| 1.3 | 1 |
| 1.5 | 2 |
| 2.3 | 2 |
| 2.5 | 2 |
| 2.7 | 3 |
| 3.5 | 4 |
| -1.5 | -2 |
| -2.5 | -2 |

## 4.6 Caracteres de declaración de tipos

Para los números con puntos flotantes, el signo de exclamación (!) y el signo de almohadilla (#) están permitidos como caracteres de declaración de tipos, pero únicamente si sustituyen a un ".0" en los literales numéricos con puntos flotantes.

Por ejemplo, para el número con el punto flotante 12.0, todas las representaciones que se incluyen a continuación son equivalentes:

```
12.0
12!
12#
```

En Visual Basic y en otros dialectos de Basic, (!) significa precisión sencilla, y (#) significa precisión doble. BasicX trata ambos como precisión sencilla, aunque podría cambiar a precisión doble si se añade al lenguaje.

Los literales numéricos hexadecimales dentro del rango &H8000 a &HFFFF (desde 32 768 a 65 535) deben tener necesariamente caracteres de declaración de tipos &.

No es legal adjuntar caracteres de declaración de tipos a nombres variables, o a literales numéricos con partes fraccionarias. Por ejemplo:

```
' Todos estos ejemplos son ilegales:
X!
X#
12.5!
12.5#
I&
255&
```

## 4.7 Matrices

Las matrices pueden ser declaras para todos los tipos de datos excepto las cadenas y otras matrices.

Ejemplo de sintaxis:

```
Dim I (1 to 3) como Entero, J (-5 to 10, 2 to 3) como Booleano

Dim X (1 To 2, 3 To 5, 5 To 6, 1 To 2, 1 To 2, 1 To 2, _
   1 To 2, -5 To -4) as Single
```

Las matrices pueden tener de1 a 8 dimensiones, y los límites inferior y superior de cada índice deben estar declarados.

Si desea pasar una matriz como un argumento a un subprograma, la matriz debe ser de una sola dimensión y tener un límite inferior de 1. Por ejemplo:

```
Dim I(1 To 5)         ' Puede pasarse a un subprograma
Dim J(0 To 5)         ' No puede pasarse – el límite inferior no es uno
Dim K(1 To 2, 1 To 3) ' No puede pasarse – la matriz no es 1D
```

La memoria disponible de una matriz está limitada por el tamaño de la memoria RAM o EEPROM. En otras palabras, una matriz no puede ser mayor que la memoria física de del sistema. Así mismo, hay un límite superior de 32 KB para la memoria máxima utilizada por una matriz. Esto se traduce en elementos de 32 K para matrices de tipo Boolean y Byte, elementos de 16 K para matrices de tipo Integer, y elementos de 8 K para matrices de tipo Long y Single.

**Advertencia** – no hay comprobaciones ni de tiempo de compilación ni tiempo de ejecución para los desbordamientos de los índices de las matrices. Si tiene lugar un desbordamiento del índice, los resultados no están definidos.

## 4.8 Variables persistentes

Las variables persistentes están almacenadas en la memoria EEPROM. Las variables retienen sus valores después de que se desconecte la alimentación. De cierta manera, las variables persistentes se comportan como si estuvieran escritas en un disco duro.

Las variables persistentes deben estar declaradas en el nivel de módulo y no está permitido que sean variables locales. Sintaxis de declaración:

```
[Public | Privad | Dim] variable As New persistent_type
```

En la que Where persistent_type es:

```
PersistentBoolean|PersistentByte|PersistentInteger|
  PersistentLong|PersistentSingle
```

Ejemplo:

```
Public I As New PersistentInteger, B As New PersistentByte
Privad X As New PersistentSingle, BL As New PersistentBoolean
Dim L As New PersistentLong
```

El orden de las variables persistentes en la memoria EEPROM es importante si desea que la ubicación de cada variable sea la misma después de cada encendido/apagado del sistema. De lo contrario, es posible que se intercambien dos o más variables sin que el programa lo sepa.

Para poder garantizar el orden de las variables persistentes, se deberían seguir 3 reglas:

1. Todas las variables persistentes deberían estar declaradas en un módulo.
2. El orden de las declaraciones de variables persistentes debe coincidir con el orden en el que se accedieron por primera vez en el tiempo de ejecución. En este contexto, "acceder" es sinónimo de operación de lectura o escritura.
3. Todas las variables persistentes deberían ser privadas.

Ejemplos:

```
Option Explicit

Private I As New PersistentInteger, J As New PersistentInteger

Private Sub Main()

  Call Init

End Sub
```

```
Private Sub Init()

   Dim K As Integer

   ' Dummy reads.
   K = I
   K = J

End Sub
```

En este ejemplo, el procedimiento Init debería llamarse antes de utilizar cualquier tipo de variables persistentes I y J. Si se invierte el orden de los comandos de asignación en Init sin invertir también el orden en el que I y J están declaradas, el orden de la memoria EEPROM será indefinido.

El orden también indefinido si declara las variables persistentes en más de un módulo.

# 5. Expresiones

## 5.1 Información general

BasicX utiliza un tipo de escritura dura, lo que significa que los operadores binarios deben operar en los tipos equivalentes. No está permitido el uso de la aritmética de modo mixto, ambas partes de un comando de asignación deben ser del mismo tipo, y cada argumento transferido a un subprograma debe tener el tipo correcto. Por ejemplo:

```
Dim I As Integer, X As Single
I = X ' Illegal -- type mismatch
```

El comando de asignación anterior es ilegal dado que I y X son de tipos diferentes. El comando podría hacerse legal utilizando una conversión explícita de tipos:

```
I = CInt(X)
```

Los literales numéricos están clasificados como universal real o universal integer. Un punto decimal en un literal numérico lo convierte en un universal real. De otro modo, el número se trataría como un entero universal (*universal integer*).

Ejemplos:

```
Dim B As Byte, I As Integer, X As Single, BB As Boolean, L As Long

B = 5         ‘ Legal
I = 5         ‘ Legal

I = 32768     ‘ Illegal -- overflow
I = 1.5       ‘ Illegal -- type mismatch
I = CInt(1.5) ‘ Legal after explicit type conversion

L = CLng(I)   ‘ Legal after explicit type conversion

X = 5.0       ‘ Legal
```

```
X = 5        ‘ Illegal -- type mismatch
X = CSng(5)   ‘ Legal after explicit type conversion
X = 5.       ‘ Illegal (missing zero after decimal point)
X = .5       ‘ Illegal (missing zero in front of decimal point)

BB = True     ‘ Legal
BB = 0       ‘ Illegal – type mismatch
```

**Problemas conocidos**

En expresiones, el compilador tiende a ser excesivamente permisivo al permitir los literales que sobrepasan los límites del rango. Por ejemplo, las expresiones Byte pueden tener literales numéricos en un rango de -32 768 a 32 767 en vez del rango correcto de 0 a 255.

Así mismo, los valores incorrectos se generan para las constantes de puntos flotantes con valores absolutos que son extremadamente pequeños – en el orden de 1.5E-45.

## 5.2 Operadores relacionales

| | |
|---|---|
| Equality (Igualdad) | = |
| Inequality (Desigualdad) | <> |
| Less (Menor que) | < |
| Greater (Mayor que) | > |
| Less or equal (Menor o igual a) | <= |
| Greater or equal (Mayor o igual a) | >= |

Los operadores relacionales dan lugar a un tipo booleano.

Los operadores de igualdad y desigualdad requieren operadores de tipos booleanos o numéricos. Los otros operadores requieren todos tipos numéricos.

## 5.3 Operadores lógicos

Y
Or
Not
Xor

Los operadores lógicos requieren operadores de tipo booleano o tipos discretos sin signo (Byte, UnsignedInteger o UnsignedLong), y que el tipo resultante coincida con el de los operadores.

Cuando las operaciones lógicas utilizan tipos numéricos como operadores, se ejecutan las operaciones de tipo bitwise. En otras palabras, las operaciones se realizan en bits individuales.

## 5.4 Operadores aritméticos

| | |
|---|---|
| Addition (suma) | + |
| Subtraction (resta) | - |
| Multiplication (multiplicación) | * |
| Divide (float) (división) | / |
| Divide (integer) (división) | \ |

Modulus (Módulos) Mod
Absolute value (Valor absoluto) Abs

Las operaciones aritméticas requieren operadores numéricos. Tenga en cuenta que las operaciones de división utilizan símbolos de operadores independientes para los operadores de punto decimal flotante y operadores discretos.

**Advertencia** – no hay comprobaciones ni de tiempo de ejecución para los desbordamientos numéricos. Si tiene lugar un desbordamiento, los resultados no están definidos.

## 5.5 Operadores de cadenas

Concatenación &

Las cadenas pueden concatenarse. Generalmente, si la cadena final es más larga que la cadena resultante, la cadena se justificará a la izquierda y se dejará en blanco. Si la cadena final es más corta, se truncará la misma.

## 5.6 Prioridad de los operadores

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| (Más alto) | [1] | Abs | Not | | | | | |
| | [2] | * | \ | / | Mod | Y | | |
| | [3] | + | - | Or | Xor | | | |
| (Más bajo) | [4] | = | > | < | <> | <= | >= | |

## 5.7 Comandos de asignación

```
i = expression
```

Los tipos de ambas partes de una asignación deben coincidir una con la otra. No se realizan conversiones implícitas de tipos.

Ejemplos:

```
Dim I como Entero (Integer), A como Sencillo (Single), X como Sencillo (Single), J como Entero (Integer)
Const Pi como Sencillo (Single) = 3.14159265

I = 3 + CInt(Log(145.0))

A = Sin(Pi/2.0) + Pi

X = CSng(I) / 3.0

J = CInt(X) \ 3  ' Tenga en cuenta la división del entero
```

# 6. Tipos sin signo

### 6.1 Información general

Se proporcionan los siguientes tipos de enteros sin signo:

| Tipo | Almacenamiento | Rango |
|---|---|---|
| Byte | 8 bits | 0 .. 255 |
| UnsignedInteger | 16 bits | 0 .. 65 535 |
| UnsignedLong | 32 bits | 0 .. 4 294 967 295 |

Los tipos UnsignedInteger y UnsignedLong realmente se tratan como clases, lo que significa que la sintaxis para la declaración de variables (en realidad objetos) es ligeramente diferente a otros tipos como Byte, Integer y Long.

Estas son las reglas para los objetos UnsignedInteger y UnsignedLong:

1. Si desea declarar objetos sin signo (*unsigned)* como variables locales o de nivel de módulo, deberá utilizar la palabra clave *New*. Ejemplos:

```
Dim I As New UnsignedInteger, L As New UnsignedLong
Public ui As New UnsignedInteger, j As New UnsignedLong
```

Sin embargo, la palabra clave *New* no es necesaria en las listas de parámetros de los subprogramas:

```
Privad Sub S(ByRef I As UnsignedInteger)
```

2. Las funciones que devuelve los objetos sin signo deben tener un comando de tipo *Set* como primera línea de la función.

Ejemplo:

```
Función F() As UnsignedInteger
Set F = New UnsignedInteger
   F = 65535
End Function
```

3. Los objetos sin signo no pueden utilizarse en comandos de tipo *Const.*.

4. Si pasa un objeto sin signo por valor (*by value*), se tratará al objeto como si estuviera protegido contra escritura (*write-protected*) dentro del subprograma llamado. Esto significa que no se puede asignar un objeto, utilizarlo como contador de bucle, o pasarlo por referencia a otro subprograma.

Un número determinado de llamadas del sistema operativo que utilizan argumentos enteros realmente tratan los argumentos como tipos sin signo. Normalmente, estas llamadas requieren ciertos retrasos de tiempo, pies I/O y comunicaciones. Las llamadas están generalmente sobrecargadas, lo que implica que puede utilizar tanto tipos son signo como tipos sin signo. Para llamadas del sistema, la ventaja de los tipos sin signo es que permiten un acceso a un amplio rango de valores.

## 6.2 Conversiones de tipos:

| | |
|---|---|
| CuInt | Convierte cualquier tipo discreto a UnsignedInteger (entero sin signo) |
| CuLng | Convierte cualquier tipo discreto a UnsignedLong (largo sin signo) |
| FixUI | Trunca los tipos de puntos flotantes, y los convierte a UnsignedInteger (entero sin signo) |
| FixUL | Trunca los tipos de puntos flotantes, y los convierte a UnsignedLong (largo sin signo) |

**Problemas comunes:**

1. Las siguientes operaciones aritméticas no están permitidas con los tipos largos sin signo (UnsignedLong):

Multiply
Divide
Mod

2. Problemas de portabilidad – si se utiliza un tipo UnsignedInteger (entero sin signo) o UnsignedLong (largo sin signo) se usa como un parámetro formal, y si el objeto se pasa por valor, se supone que el parámetro real está restringido a un solo objeto. BasicX de manera errónea permite que los literales numéricos y expresiones sean parámetros reales.

# 7. Reglas sintácticas estrictas frente a permisivas

## 7.1 Opciones de compilador

El compilador puede configurarse para que utilice tanto reglas sintácticas permisivas como estrictas. Esta opción afecta al tratamiento de los literales numéricos, expresiones lógicas y contadores de bucle For-Next. Por defecto, la configuración es el uso de reglas estrictas.

## 7.2 Reglas permisivas

Si prefiere reglas más permisivas, puede desactivar el control de sintaxis estrictas, que tendrá los siguientes efectos:

- Para contadores de bucle For-Next:
  - No es necesario que los contadores sean variables locales. Pueden ser variables globales, por ejemplo.
  - Los contadores no están protegidos contra escritura dentro de los bucles, lo que implica que podrá modificarlos dentro de los bucles, y pasarlos a otros subprogramas por referencia.
  - El rango de un contador no está restringido a su bucle. Podrá tener el código que depende del valor de un contador después de un bucle.
  - Los literales numéricos y operaciones lógicas:
  - Los tipos discretos sin signo (Integer y Long) pueden aparecer en expresiones de tipo *bitwise-logical*.

- Dispone de una selección más amplia de tipos de caracteres de declaración que pueden añadirse a los literales numéricos hexadecimales. Puede utilizar un signo & (ampersand) o caracteres de porcentaje, o ningún carácter.

**Nota:** La elección entre reglas sintácticas permisivas o estrictas no afecta en gran medida en el modo de compilación del programa BasicX en un entorno Visual Basic. Ambas convenciones de códigos tiene la misma compatibilidad ascendente.

Tenga en cuenta que si escribe una biblioteca para distribuirla en forma de código fuente, deberá asegurarse que la biblioteca compila utilizando las reglas sintácticas estrictas. Esto implica que los usuarios de la biblioteca tendrán más flexibilidad para su propio código –pueden elegir entre las reglas estrictas o permisivas. De otro modo los usuarios están obligados a utilizar el modo permisivo.

**Problemas comunes:**

En el modo permisivo, algunos literales numéricos hexadecimales arrojan valores incorrectos para los tipos largos sin signos (UnsignedLong). Por ejemplo, si X es un tipo UnsignedLong, la asignación X = &HFFFFFFFF fija X en 65 535 en lugar del correcto 4 294 967 295.

Se utiliza un parche para activar el control sintáctico estricto.

# 8. Comandos Varios

## 8.1 Comandos de atributos

Se ignoran los comandos **Attribute Visual Basic_Name**. Todos los demás comandos de atributos son ilegales.

Ejemplo de un comando de atributo legal:

```
Attribute Visual Basic_Name = "Módulo1"
```

(En Visual Basic, los nombres de los módulos se toman del atributo Visual Basic_Name. Al contrario, BasicX deriva los nombres de los módulos directamente de los nombres de los ficheros de los módulos.)

## 8.2 Comando de opciones

La **Option Explicit** requiere que las variables se declaren antes de su uso, por defecto en BasicX. Cualquier otro comando de opción es ilegal.

Sintaxis:

```
Option Explicit
```

### 8.3 Comando WITH

Un comando **WITH** le permite utilizar identificadores cortos para los objetos, lo que implica que puede omitir el cualificador del nombre del objeto en la referencia del objeto. Actualmente, estos comandos puede sólo utilizarse con objetos de tipo Register (consulte el documento de Referencia de Sistema Operativo - *Operating System Reference* – si desea obtener una explicación de los objetos Register). No es posible utilizar otros tipos de objetos con los comandos **WITH**.

Un comando **WITH** puede sólo utilizarse dentro de un subprograma. Además un comando **WITH** que precede a un bloque de código debe estar cerrado por un comando **END WITH** al final del bloque, pero antes del final del subprograma. No está permitido anidar comandos **WITH**.

Sintaxis:

```
With Register
   [statements]
End With
```

Código de ejemplo:

```
' Los siguientes dos comandos de asignación son equivalentes.
Register.OCR1AH = 255
With Register
   .OCR1AH = 255
End With
```

## 9. Palabras clave restringidas

Las palabras clave restringidas están reservadas por el idioma. No está permitido utilizar estas palabras como identificadores definidos por el usuario para nombres de variables, de subprograma o entidades similares:

| | | | |
|---|---|---|---|
| abs | cstr | else | let |
| y | currency | elseif | like |
| matriz | cvar | end | lock |
| as | cverr | eqv | long |
| attribute | date | erase | loop |
| boolean | defbool | exit | lset |
| byref | defbyte | false | me |
| byte | defcur | fix | mod |
| byval | defdate | for | new |
| call | defdbl | function | next |
| case | defdec | get | not |
| cbool | defint | gosub | on |
| cbyte | deflng | goto | open |
| ccur | defobj | if | option |
| cdate | defsng | imp | or |
| cdbl | defstr | in | preserve |
| cdec | defvar | input | print |
| cint | dim | int | privad |
| clng | do | integer | public |
| close | doevents | is | put |
| const | double | lbound | redim |
| csng | each | len | rem |

| | | | |
|---|---|---|---|
| resume | single | then | vb_name |
| return | spc | to | wend |
| rset | static | true | while |
| seek | stop | ubound | with |
| select | string | unlock | write |
| set | sub | until | xor |
| sgn | tab | variant | |

# 10. Preguntas y respuestas más frecuentes sobre BASIC X

**1. Pregunta:** ¿Por qué está permitido el uso del tipo de caracteres de declaración (!) y (#) para los literales numéricos de puntos flotantes?

**Respuesta:** Estos caracteres facilitan la creación de código en el entorno Visual Basic. El problema es que Visual Basic no le permitirá escribir punto-cero después de un número. Por ejemplo, el número 12.0 aparecerá como "12!" o "12#" en Visual Basic, y sin intenta escribir "12.0", Visual Basic lo sustituirá automáticamente por "12#" (! significa precisión sencilla, # significa precisión doble).

Por otro lado, Visual Basic aceptará el estilo punto-cero si este número ha sido creado en un fichero externo a Visual Basic siempre que no intente editarlo dentro de Visual Basic.

**2. Pregunta:** Cuando se utilizan las variables persistentes, ¿por qué tiene el orden del primer acceso durante el tiempo ejecución que coincidir con el orden de las declaraciones?

**Respuesta:** Las reglas de ordenación existen por la forma que el compilador asigna la memoria persistente. El compilador asigna libremente la memoria basándose tanto en el orden de declaraciones, o en el orden del primer acceso durante el tiempo de ejecución. La única forma de garantizar el control sobre la ordenación es asegurarse de que ambos métodos coinciden.

**3. Pregunta:** ¿Por qué no se puede utilizar una variable persistente como una variable local?

**Respuesta:** Porque la variable persistente es tal como su nombre indica -- persistente. Una variable local desaparece cuando regresa de un subprograma en el que se ha declarado la variable.

**4. Pregunta:** ¿Cómo se almacenan internamente las variables booleanas?

**Respuesta:** Internamente las variables booleanas se almacenan como un byte. Numéricamente, un valor “false” (falso) es 0 y un valor “true” (verdadero) es 255.

**5. Pregunta:** En los bucles For-Next, ¿por qué no se pueden utilizar un tamaño de paso distinto de ±1? ¿Por qué no se puede utilizar un contador de bucles de puntos flotantes?

**Respuesta:** Esta restricción es deliberada para los bucles For-Next. Los tamaños de pasos arbitrarios frecuentemente dificultan que los programadores puedan determinar exactamente cuándo un bucle llega a su fin. Los contadores de bucles de puntos flotantes pueden frecuentemente provocar problemas similares debido a errores acumulados de redondeo.

**6. Pregunta:** Cuando se utiliza ByVal para pasar una variable de entero sin signo (UnsignedInteger) o largo sin signo (UnsignedLong) a un subprograma, ¿por qué está protegida contra escritura la variable dentro del subprograma llamado? ¿Por qué no se puede asignarla a la copia como otras variables?

**Respuesta:** Se debe a cuestiones de compatibilidad. Visual Basic trata estas entidades como objetos, y aparentemente no hay distinción entre ByVal y ByRef para un objeto en Visual Basic – ambos son equivalentes a ByRef. Para mantener la compatibilidad ascendente con Visual Basic, BasicX implementa ByVal aunque prohíba la asignación de las propiedades del objeto.

# 11. Cuestiones de portabilidad

Esta sección trata diversas dudas que surgen si desea conectar el programa Basic Express a un compilador Visual Basic en un PC. En este apartado se tratarán únicamente temas relacionados con el lenguaje.

Las cuestiones relacionadas con las diferencias existentes entre los sistemas operativos no se cubrirán en este apartado. Por ejemplo, los programas Visual Basic normalmente dan por supuesto la existencia de un entorno informático Windows que incluye un monitor, teclado y ratón. Ninguno de estos dispositivos están necesariamente presente en un sistema integrado como BasicX, lo que implica que el programa Visual Basic integrado controla los botones de comandos; no obstante las barras deslizadoras y otros objetos relacionados con las ventanas no están controlados.

Las dependencias de otros sistemas operativos como las entradas/salidas serie, están dentro de la misma categoría.

### Comprobación de desbordamiento de la memoria

**BasicX:** No comprueba el desbordamiento de la pila.

**Visual Basic:** Comprueba el desbordamiento de la pila.

### Inicialización automática de variables

**BasicX:** No inicializa automáticamente las variables.

**Visual Basic:** Inicializa automáticamente las variables. Las variables numéricas, por ejemplo, se inicializan a cero.

### Constantes binarias

**BasicX:** Incluye las constantes binarias definidas por el sistema, tales como bx11110000.

**Visual Basic:** Las constantes binarias no están integradas, aunque pueden simularse mediante la declaración de las siguientes constantes simbólicas en el código del nivel de módulo:

```
Public Const bx00000000 As Byte = &H00
Public Const bx00000001 As Byte = &H01
Public Const bx00000010 As Byte = &H02
```

```
[...]
Public Const bx11111111 As Byte = &HFF
```

**Comandos de atributos Visual Basic_Name**

**BasicX:** Ignora los comandos de atributos Visual Basic_Name

**Visual Basic:** Usa el atributo Visual Basic_Name para definir el nombre del módulo en el que aparece el atributo. Tenga en cuenta que los comandos de atributos están controlados automáticamente por Visual Basic – los comandos no son normalmente visibles en el código fuente al utilizar un entorno Visual Basic para la edición.

**Comandos de cadena – longitud fija frente a Longitud variable**

**BasicX:** Cuando se pasa una variable de cadena como un comando a u subprograma, retiene su identidad como una cadena de longitud fija o variable. Es decir, las cadenas de longitud fija siguen siendo fijas, y las cadenas de longitud variable siguen siendo variables.

**Visual Basic:** Convierte todas las comandos de cadena a longitud variable dentro del subprograma llamado. Sólo después del retorno la cadena de longitud fija recuperará su atributo de longitud fija.

## *Cadenas limitadas*

**BasicX:** Permite comandos de tipo medio (Mid), tales como Mid(S, 1, 2) = *expresión*, en la que S es una cadena limitada.

**Visual Basic:** No permite comandos de tipo medio (Mid) como cadenas limitadas. Las funciones Mid sin embargo sí están permitidas.

**Tipos UnsignedInteger (enteros sin signo) y UnsignedLong (largo sin signo)**

**BasicX:** Estos tipos los proporciona el sistema.

**Visual Basic:** Estos tipos no los proporciona el sistema, aunque pueden implementarse como clases. Así mismo, Visual Basic no proporciona los tipos de enteros que coinciden exactamente con el rango de enteros sin signo desde 0 hasta 65. 535 (UnsignedInteger) o largo sin signo desde 0 hasta 4 millones (UnsignedLong). En la mayoría de los casos el tipo largo de 32-bit de Visual Basic es lo más cercano a estos objetos, aunque los tipos "Decimal" o “Currency” de Visual Basic pueden utilizarse para el tipo largo sin signo (UnsignedLong) si no necesita utilizarlo para operaciones bit de tipo booleano en el tipo.

**Operador Mod**

**BasicX:** Devuelve un valor positivo o cero independientemente de los signos de los operadores. Por ejemplo:

```
(A Mod B) se trata como un equivalente a  (Abs(A) Mod Abs(B))
```

**Visual Basic:** Puede devolver un resultado negativo dependiendo de los signos de los operadores.

### Notación científica y tipos de datos

**BasicX:** Si un literal numérico incluye una notación científica, o si no se incluyen ni el punto decimal ni un carácter de declaración de tipos, BasicX trata al número como un tipo discreto.

**Visual Basic:** Trata un literal numérico como un tipo de punto flotante si se utiliza una notación científica, independientemente de si está o no presente un punto decimal.

### Comprobación de error general

Si desea conectar un programa desde BasicX a Visual Basic, es posible minimizar las diferencias de ejecución seleccionando las siguientes opciones del compilador de Visual Basic:

- Quitar las casillas de control de Matriz Bounds Checks
- Quitar las casillas de control de comprobaciones de desbordamiento de enteros (Integer Overflow Checks)
- Quitar las casillas de control de comprobaciones de errores de puntos flotantes (Floating Point Error Checks)

Para acceder a estas opciones en Visual Basic, vaya al menú Project, seleccione Properties. Haga clic en la pestaña Compile, y pulse el botón Advanced Optimizations (Optimizaciones avanzadas). Deberían aparecer las selecciones realizadas.