Basic Express
Language Reference

Version 2.0

Contents

General

Modules

Modules allow you to split a program into multiple files. By using modules, you can control the visibility of subprograms, which can be public (global) or private to a module. Variables and constants can also be global, local to modules, or local to subprograms.

A public subprogram can be called from other modules. A private subprogram can be called only from within the module in which it appears.

Public and private variables appearing in *module level* code have similar properties, as do public and private constants. Module level code refers to code that doesn't belong to any subprograms, and appears at the beginning of a module.

Example of a module:

```
Public A As Integer
Private B As Single ' Module level code ends here.

Public Sub Main()

    Dim K As Integer

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

End Sub

Private Sub Square(X As Single)

    X = X * X

End Sub
```

In this example, variable A and procedure Main are visible in all other modules. Variable B and procedure Square are visible only from within this module. Variable K is a local variable visible only within procedure Main.

Variables in module level code also remember their values between subprogram calls. These variables are sometimes called *static* variables. By contrast, local variables like K do not retain their values between calls.

Module names are taken from filenames, which means the filenames (minus extensions) must be legal Basic identifiers.

Identifiers -- All BasicX identifiers must start with a letter, and all other characters must be letters, digits or underscores. Identifiers are case-insensitive. An identifier can be up to 255 characters long, and all characters are significant.

Main program

The program starts execution with a procedure called Main, which must be a public procedure.

You can have more than one *private* procedure called Main, as long as you don't have more than one Main per module. A single *public* Main is required, however -- otherwise the program won't know where to start.

Statement format

To make a long statement easier to read, it can be extended between two or more lines by adding a line continuation character (underscore) as the last character on a line.

The underscore has to be preceded by one or more blanks or tabs, and it must be the last character on a line. Nothing, including comments, can follow on the same line. Comments in particular are not allowed to be extended in this manner.

Example:

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

String literals also can't be extended in this manner, although you can use concatenation operators to extend a string to the next line. Both lines in the following example are equivalent:

```
S = "Hello, world"

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

Comment format

An apostrophe is used to denote comments. Example:

```
I = 32767 ' Comment
```

All characters to the right of the apostrophe are ignored.

Subprograms

General

A subprogram allows you to take a group of related statements and treat them as a single unit.

Subprograms consist of procedures and functions. The difference between a procedure and function is that a function can appear as part of an expression, but a procedure must be called in a standalone statement.

There is no specific limit to the nesting level of subprogram calls -- the only limit is available RAM. Subprogram calls are also allowed to be recursive.

Sub procedures

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

You call a procedure by using the *Call* keyword. You can exit a procedure by using an Exit Sub statement. Example:

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

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

End Sub
```

When you call a procedure, the *Call* keyword is optional. If Call is omitted, the parentheses around the actual parameters (if any) must also be omitted. For example, the following 2 statements are equivalent:

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

Functions

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

A function is a subprogram that is similar to a procedure. The difference is that you call a function by using its name in an expression, followed by the function's argument list (if any) in parentheses.

The function returns a value, which can be defined by assigning to the function name inside the function itself. In other words, you can treat the name as if it were a local variable, and you can usually read and write to the name inside the function. One exception is for string functions, in which the function return is write-only inside the function.

Example:

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

    F = 2 * i   ' This defines the function return.

    F = F * F   ' You can also read from the function name.

End Function
```

You can also exit a function by using an Exit Function statement. Example:

```
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
```

Function return types

Functions can return non-persistent scalar types or string types.

Example syntax for string functions:

```
Function F() As String
    F = "Hello, world"  ' F is write-only.
End Function
```

For efficiency reasons, a string function return is write-only. That is, you can assign to the function return, but you can't read it, use it in an expression or pass it to another subprogram. Every assignment to the function return must be immediately followed by an "Exit Function" or "End Function" statement.

If a function returns an UnsignedInteger or UnsignedLong object, the first statement in the function must be a *Set* statement. Example:

```
Function F() As UnsignedInteger
    Set F = New UnsignedInteger
    [...]
End Function
```

Passing parameters to subprograms

Parameters can be passed to a subprogram by reference (ByRef) or by value (ByVal).

Pass by reference -- if you pass a parameter by reference, any changes to the parameter will propogate back to the caller. Pass by reference is the default.

Pass by value -- if you pass a parameter by value, no changes are allowed to propogate back to the caller. With a few exceptions, if an argument is passed by value, a copy is made of the argument, and the called subprogram operates on the copy. Changes made to the copy have no effect on the caller.

One exception is for string parameters passed by value. For efficiency reasons, a copy of the string is not made. Instead, the string is write-protected in the called subprogram, which means you can neither assign to it nor pass it by reference to another subprogram. You are allowed to pass it *by value* to another subprogram, however.

The other exception is for types UnsignedInteger and UnsignedLong, which are treated similarly -- these parameters are write-protected in called subprograms.

Actual vs. formal parameters -- the type and number of the actual parameters must match that of the "formal" parameters (the formal parameters appear in the subprogram declaration). If there is a type mismatch, the compiler will declare an error. It will not do implicit type conversions.

Example syntax:

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

In the following example, SortList is an array:

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

Note that string S is passed by reference, which is the default.

Restrictions on passing mechanisms:

1. Scalar variables and array elements can be passed by value or by reference.

2. An array can be passed by reference only. The array must also be one-dimensional and have a lower bound of one. Any other kind of array can't be passed at all, regardless of the passing mechanism.

3. Numeric expressions and numeric literals can be passed by value but not by reference. The same applies to boolean expressions and boolean literals.

4. Persistent variables can be passed by value only, and only after they've been converted to the corresponding non-persistent type.*

5. If you use pass by value for types String, UnsignedInteger and UnsignedLong, the parameters are write-protected inside the called subprograms, which means you're not allowed to change their values or use them as For-Next loop counters. If you pass these parameters to another subprogram, they must be passed by value, not by reference.

6. Functions are not allowed to accept general string expressions, including string literals, as actual parameters. If you pass a string to a function, the string must be a variable, even if it's passed by value. (This restriction does not apply to procedures.)

* Although BasicX generally does not do implicit type conversions, it will automatically convert a persistent variable to the corresponding non-persistent type when the variable is passed as a subprogram argument. For example, a PersistentInteger variable is converted to Integer.

To summarize:

Parameter	ByRef	ByVal
Scalar variable	Yes	Yes
Array element	Yes	Yes
1D array, lower bound = 1	Yes	No
Multidimensional array	No	No
Array with lower bound not 1	No	No
Numeric expression	No	Yes
Numeric literal	No	Yes
Boolean expression	No	Yes
Boolean literal	No	Yes
Persistent variable	No	Yes

Control structures

If-Then statement

```
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
```

Examples:

```
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
```

Do-Loop Statement

Infinite loop:

```
Do
     [statements]
Loop
```

You can also add a "While" qualifier, which tests a boolean expression to determine whether to execute the body of the loop. The loop continues as long as the expression is true. The test can be at the beginning or end of the loop. If the test is at the end, the body is always executed at least once.

The "Until" qualifier is similar to "While", except the logic is reversed. That is, the loop is *terminated* rather than continued when the boolean expression is true.

Examples:

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

Do
    [statements]
Loop While (boolean_expression)

Do Until (boolean_expression)
    [statements]
Loop

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

Exit Do can be used to exit any of the Do-Loops. Do-Loops can be nested up to a level of ten.

Examples:

```
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)
```

For-Next Statement

A For-Next statement can be used to execute a loop a specific number of times.

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

The loop runs from start_value to end_value in steps of +1 or -1. If this gives a zero or negative count, the loop is not executed at all.

Loop_counter must be a discrete type. Also, loop_counter, start_value and end_value must all be the same type. Both start_value and end_value can be expressions.

Loop counters must be local variables, and you are not allowed to change a counter inside the loop. In other words, a counter is treated as if it were a constant within the loop. If you want to pass the counter to a subprogram, you must pass it by value. Pass by reference is not allowed -- otherwise the called subprogram could change the counter indirectly.

At the end of a For-Next loop, the counter is assumed to be undefined. Accordingly, the scope of each counter is limited to the loop to which it belongs, with the exception that multiple non-nested loops are allowed to share the same counters.

In other words, you can't use loop counters outside of their loops.

"Exit For" can be used to exit a For-Next loop.

For-Next loops can be nested up to a level of ten.

Examples:

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

This loop is executed ten times, with i being incremented each time.

The next example illustrates an Exit-For statement. The loop counter is decremented, and the maximum number of iterations is ten.

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

Select-Case Statement

A Select-Case statement can be used to select between a list of alternatives. Syntax:

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

The test_expression is evaluated once, at the top of the loop. The program then branches to the first expression_list that matches the value of test_expression, and the associated block of statements is executed.

If no match is found, the program branches to the optional Case Else statement, if present. If there is no Case Else, the program branches to the statement following the End Select statement.

The test_expression must be a discrete, non-string type (boolean or discrete numeric). Each expression_list choice must have the same type as test_expression.

Example:

```
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
```

GoTo Statement

A GoTo branches unconditionally to the specified label. Example:

```
GoTo label_name
```

```
label_name:
```

Labels must be followed by a colon.

Variables, constants and data types

Data types

Type	Storage	Range
Boolean	8 bits	True .. False
Byte	8 bits	0 .. 255
Integer	16 bits	-32 768 .. 32 767
Long	32 bits	-2 147 483 648 .. 2 147 483 647
Single	32 bits	-3.402 823 E+38 .. 3.402 823 E+38
String	Varies	0 to 64 characters
BoundedString	Varies	0 to 64 characters

String storage -- there are 3 kinds of strings -- variable-length, fixed-length, and bounded. The maximum allowable length of any of the strings depends on the compiler setting. The selectable range is 1 to 64 characters, with a default of 64.

The storage required for a fixed-length string is 2 bytes plus the number of characters specified in the declaration, up to the user-defined maximum. A variable length string always requires a constant storage, which is 2 bytes plus the user-defined maximum. A bounded string requires the declared length plus 2 bytes for storage.

In this example, it is assumed that the compiler is set to a 45 character limit:

```
' Fixed-length string, requires 2 + 5 = 7 bytes of storage.
Dim A As String * 5

' Variable-length string, requires 2 + 45 = 47 bytes of storage.
Dim B As String
```

Bounded strings – syntax is as follows:

```
Dim B As BoundedString_N  ' Where N is the maximum length.
```

Examples:

```
Dim A As BoundedString_10  ' Allows up to 10 characters.
Dim A As BoundedString_64  ' Allows up to 64 characters.
```

Declarations

All variables must be declared before they're used. Implicit declarations are not allowed.

For variables declared in module-level code:

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

15

Public variables are global and visible throughout the entire program. Private variables are visible only in the module in which they appear. The default is private.

For variables declared inside a subprogram:

```
Dim variable As type
```

Variables declared inside a subprogram are visible only inside the subprogram. Examples:

```
Public Distance As Integer    ' Module-level variable, global
Private Temperature As Single ' Module-level variable, local to module

Sub ReadPin()
    Dim PinNumber As Byte     ' Variable is local to this subprogram
End Sub
```

Strings can have a fixed or variable length. Example syntax:

```
Dim S1 As String      ' Variable length
Dim S2 As String * 1  ' 1-character string
Dim S2 As String * 64 ' 64-character string
```

A variable length string can hold 0 to 64 characters, depending on the compiler setting.

Constants

For constants declared in module-level code:

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

The default is private.

For constants declared inside a subprogram:

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

Examples:

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

Numeric literals

Decimal integer examples:

```
1
-1
10
255
```

Note that integer numeric literals are not allowed to have a decimal point.

Decimal floating point examples:

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

Hexadecimal integer examples:

```
&H3
&HFF
&H7FFF      ' 32767
-&H8000&    ' -32768 (note trailing ampersand)
```

Trailing ampersands are required for hexadecimal numbers in range &H8000 to &HFFFF (32 768 to 65 535).

Binary constants:

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

Binary constants have a "bx" prefix followed by an 8-digit binary number. Strictly speaking, these entities are not numeric literals, but are treated as equivalent to symbolic constants of type Byte.

Converting data types

Function	Result
CBool	Boolean
CByte	Byte
CInt	Integer
CLng	Long
CSng	Single
CStr	String
FixB	Byte
FixI	Integer
FixL	Long

CBool allows only a Byte type as an operand.

FixB, FixI and FixL allow only floating point types as operands. These three functions use truncation to convert to discrete types (see also Fix, which is similar but returns a floating point value).

The other functions use any numeric types as operands, but with the following difference -- when CByte, CInt and CLng are used with floating point operands, "statistical rounding" is used. This means the number is converted to the nearest integer, unless the number is exactly halfway between two integers, in which case the nearest *even* integer is returned. Examples:

```
  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
```

Type declaration characters

For floating point numbers, the exclamation point (!) and pound sign (#) are allowable as type declaration characters, but only if they replace a trailing ".0" in floating point numeric literals.

For example, for the floating point number 12.0, all of the following representations are equivalent:

```
12.0
12!
12#
```

In VB and other Basic dialects, (!) signifies single precision, and (#) signifies double precision. BasicX treats both as single precision, although this may change if double precision is added to the language.

Hexadecimal numeric literals in range &H8000 to &HFFFF (32 768 to 65 535) are required to have ampersand type declaration characters.

It is illegal to append type declaration characters to variable names, or to numeric literals with fractional parts. For example:

```
' All of these are illegal:
X!
X#
12.5!
12.5#
I&
255&
```

Arrays

Arrays can be declared for all data types except strings or other arrays. Example syntax:

```
Dim I(1 to 3) As Integer, J(-5 to 10, 2 to 3) As Boolean

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
```

Arrays can have 1 to 8 dimensions, and both the upper and lower bound of each index must be declared.

If you want to pass an array as an argument to a subprogram, the array must be one-dimensional and have a lower bound of 1. For example:

```
Dim I(1 To 5)          ' Can be passed to subprogram
Dim J(0 To 5)          ' Can't be passed – lower bound is not one
Dim K(1 To 2, 1 To 3)  ' Can't be passed – array is not 1D
```

The memory available for an array is limited by the RAM or EEPROM memory size. In other words, an array can't be larger than the physical memory in the system. In addition, there is a 32 KB upper limit for the total amount of memory used by an array. That translates to 32 K elements for Boolean and Byte arrays, 16 K elements for Integer arrays, and 8 K elements for Long and Single arrays.

Warning -- there are neither compile time nor runtime checks for array index overflows. If an index overflow occurs, results are undefined.

Persistent Variables

Persistent variables are stored in EEPROM memory. The variables retain their values after power is turned off. In a sense, persistent variables behave as if they are written to a solid-state disk.

Persistent variables must be declared at module level and are not allowed as local variables. Declaration syntax:

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

Where persistent_type is:

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

Example:

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

The ordering of persistent variables in EEPROM memory is important if you want the location of each variable to be the same after cycling power on and off. Otherwise, two or more variables may be interchanged without the program knowing about it.

In order to guarantee the ordering of persistent variables, 3 rules should be followed:

1. All persistent variables should be declared in one module.

2. The ordering of declarations of persistent variables must match the order in which the variables are first accessed at runtime. In this context, "accessed" means either a read or write operation.

3. All persistent variables should be private.

Examples:

```
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
```

In this example, procedure Init should be called before any other use of persistent variables I and J. If you reverse the order of assignment statements in Init without also reversing the order in which I and J are declared, the EEPROM ordering is undefined.

Ordering is also undefined if you declare persistent variables in more than one module.

Expressions

General

BasicX uses strong typing, which means binary operators must operate on equivalent types. Mixed-mode arithmetic is not allowed, both sides of an assignment statement must be of the same type, and each argument passed to a subprogram must have the correct type. For example:

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

The above assigment statement is illegal since I and X are of different types. The statement could be made legal by using an explicit type conversion:

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

Numeric literals are classified as either universal real or univeral integer. A decimal point in a numeric literal makes it universal real. Otherwise the number is treated as a universal integer.

Examples:

```
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
```

Known Bugs

In expressions, the compiler tends to be overly permissive in allowing numeric literals that exceed range constraints for the expression type. For example, Byte expressions are allowed to have numeric literals in range -32 768 to 32 767 instead of the correct range 0 to 255.

Also, incorrect values are generated for floating point constants with absolute values that are extremely small -- on the order of 1.5E-45.

Relational operators

```
Equality            =
Inequality          <>
Less                <
Greater             >
Less or equal       <=
Greater or equal    >=
```

Relational operators yield a boolean type.

The equality and inequality operators require operands of boolean or numeric types. The other operators all require numeric types.

Logical operators

```
And
Or
Not
Xor
```

Logical operators require operands of boolean type or unsigned discrete types (Byte, UnsignedInteger or UnsignedLong), and the resulting type matches that of the operands.

When logical operations use numeric types as operands, bitwise operations are done. In other words, the operations are done on individual bits.

Arithmetic operators

```
Addition            +
Subtraction         -
Multiplication      *
Divide (float)      /
Divide (integer)    \
Modulus             Mod
Absolute value      Abs
```

Arithmetic operations require numeric operands. Note that divide operations use separate operator symbols for floating point and discrete operands.

Warning -- there are no runtime checks for numeric overflow. If overflow occurs, results are undefined.

String operators

```
Concatenation       &
```

Strings can be concatenated. Generally, if the destination string is larger than the resulting string, the string is left-justified and blank-filled. If the destination string is smaller, the string is truncated.

Operator precedence

```
(Highest)    [1]    Abs    Not
             [2]    *      \      /    Mod      And
             [3]    +      -      Or   Xor
(Lowest)     [4]    =      >      <    <>       <=      >=
```

Assignment statements

```
i = expression
```

The types of both sides of an assigment statement must match. No implicit type conversions are done.
Examples:

```
Dim I As Integer, A As Single, X As Single, J As Integer
Const Pi As Single = 3.14159265

I = 3 + CInt(Log(145.0))

A = Sin(Pi/2.0) + Pi

X = CSng(I) / 3.0

J = CInt(X) \ 3   ' Note integer division
```

Unsigned types

General

The following unsigned integer types are provided:

```
Type                 Storage      Range

Byte                 8 bits       0 .. 255

UnsignedInteger      16 bits      0 .. 65 535

UnsignedLong         32 bits      0 .. 4 294 967 295
```

UnsignedInteger and UnsignedLong types are actually treated as classes, which means the syntax for declaring variables (actually objects) is slightly different from other types such as Byte, Integer and Long.

These are the rules for UnsignedInteger and UnsignedLong objects:

1. If you want to declare unsigned objects as local or module-level variables, you need to use the *New* keyword. Examples:

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

The New keyword is not required in subprogram parameter lists, however:

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

2. Functions that use unsigned object returns must have a *Set* statement as the first line of the function.

Example:

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

3. Unsigned objects can't be used in *Const* statements.

4. If you pass an unsigned object by value, the object is treated as if it were write-protected within the called subprogram. That means you can't assign to the object, use it as a loop counter, or pass it by reference to another subprogram.

A number of operating system calls that use integer arguments actually treat the arguments as unsigned types. These calls typically involve time delays, pin I/O and networking. The calls are generally overloaded, which means you can choose to use either signed or unsigned types. For system calls, the advantage to unsigned types is they allow access to the full range of values.

Type conversions:

```
CuInt    Converts any discrete type to UnsignedInteger
CuLng    Converts any discrete type to UnsignedLong
FixUI    Truncates floating point type, converts to UnsignedInteger
FixUL    Truncates floating point type, converts to UnsignedLong
```

Known Bugs:

1. The following arithmetic operations are not allowed for UnsignedLong types:

```
Multiply
Divide
Mod
```

2. Portability issue -- if an UnsignedInteger or UnsignedLong is used as a formal parameter, and if the object is passed by value, the actual parameter is supposed to be restricted to a single object. BasicX erroneously allows numeric literals and expressions as actual parameters.

Strict vs. permissive syntax rules

Compiler option

The compiler can be configured to use either strict or permissive syntax rules. This option affects how numeric literals, logical expressions and For-Next loop counters are treated. The default is to use strict rules.

Permissive rules

If you prefer more permissive rules, you can disable strict syntax checking, which has the following effects:

- For-Next loop counters:

    - Counters are not required to be local variables. They can be global variables, for example.

    - Counters are not write-protected inside loops, which means you're allowed to change them inside loops, as well as pass them to subprograms by reference.

    - The scope of a counter is not restricted to its loop. You are allowed to have code that depends on the value of a counter after a loop.

- Numeric literals and logical operations:

    - Signed discrete types (Integer and Long) are allowed to appear in bitwise-logical expressions.

    - You have a wider choice of type declaration characters that can be appended to hexadecimal numeric literals. You can use ampersand or percent characters, or no characters.

Note: The choice of strict vs. permissive syntax rules does not have much effect on whether a BasicX program will compile in a VB environment. Both coding conventions have about the same *upward* compatibility.

Note that if you write a library to be distributed in source code form, you may want to make sure the library compiles using strict syntax checking. That means users of the library have more flexibility for their own code -- they can choose strict or permissive mode. Otherwise users are forced to use permissive mode.

Known Bugs:

In permissive mode, some hexadecimal numeric literals result in incorrect values for UnsignedLong types. For example, if X is type UnsignedLong, the assignment X = &HFFFFFFFF sets X to 65 535 rather than the correct 4 294 967 295.

A workaround is to turn on strict syntax checking.

Miscellaneous statements

Attribute statement

Attribute VB_Name statements are ignored. All other attribute statements are illegal.

Example of a legal attribute statement:

```
Attribute VB_Name = "Module1"
```

(In Visual Basic, module names are taken from the VB_Name attribute. By contrast, BasicX derives module names directly from module filenames.)

Option Statement

Option Explicit requires that variables are declared before use, which is the default in BasicX. All other option statements are illegal.

Syntax:

```
Option Explicit
```

With Statement

A **With** statement allows you to use shorthand identifiers for objects, which means you can omit the object name qualifier from an object reference. Currently these statements can only be used with Register objects (see the Operating System Reference for an explanation of Register objects). No other objects are allowed in **With** statements.

A **With** statement can only be used inside a subprogram, and a **With** statement that precedes a block of code must be terminated by an **End With** statement at the end of the block, but before the end of the subprogram. Nested **With** statements are not allowed.

Syntax:

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

Example code:

```
' The following two assignment statements are equivalent.
Register.OCR1AH = 255
With Register
    .OCR1AH = 255
End With
```

Restricted keywords

Restricted keywords in BasicX

Restricted keywords are reserved by the language for its own use. These words are not allowed to be used as user-defined identifiers for variable names, subprogram names or similar entities:

abs	defcur	imp	rem
and	defdate	in	resume
array	defdbl	input	return
as	defdec	int	rset
attribute	defint	integer	seek
boolean	deflng	is	select
byref	defobj	lbound	set
byte	defsng	len	sgn
byval	defstr	let	single
call	defvar	like	spc
case	dim	lock	static
cbool	do	long	stop
cbyte	doevents	loop	string
ccur	double	lset	sub
cdate	each	me	tab
cdbl	else	mod	then
cdec	elseif	new	to
cint	end	next	true
clng	eqv	not	ubound
close	erase	on	unlock
const	exit	open	until
csng	false	option	variant
cstr	fix	or	vb_name
currency	for	preserve	wend
cvar	function	print	while
cverr	get	private	with
date	gosub	public	write
defbool	goto	put	xor
defbyte	if	redim	

Basix Express language FAQ

Frequently asked questions

1. Question: Why are the type declaration characters (!) and (#) allowed for floating point numeric literals?

Answer: They make it easier to use code created in a Visual Basic environment. The problem is that VB will not allow you to type point-zero after a number. For example, the number 12.0 is allowed to appear as "12!" or "12#" in VB, and if you try to type "12.0", VB automatically replaces it with "12#" (! means single precision, # means double precision).

VB will, on the other hand, accept the point-zero style from a file created somewhere else as long as you don't try to edit the file inside VB.

2. Question: When you use persistent variables, why does the ordering of first access at runtime have to match the ordering of declarations?

Answer: The ordering rules exist because of the way the compiler allocates persistent memory. The compiler is free to allocate memory based either on declaration ordering, or on the ordering of first access at runtime. The only way to guarantee control over ordering is to make sure both methods match.

3. Question: Why can't I use a persistent variable as a local variable?

Answer: Because a persistent variable is just that -- persistent. A local variable goes away when you return from a subprogram in which the variable was declared.

4. Question: How are Boolean variables stored internally?

Answer: Internally a Boolean variable is stored as a byte. Numerically, false is 0 and true is 255.

5. Question: In For-Next loops, why can't I use a step size other than ±1? And why can't I use a floating point loop counter?

Answer: This was a deliberate restriction on For-Next loops. Arbitrary step sizes often make it difficult for programmers to determine exactly when a loop terminates. Floating point loop counters can frequently lead to similar problems due to accumulated roundoff errors.

6. Question: When you use ByVal to pass an UnsignedInteger or UnsignedLong variable to a subprogram, why is the variable write-protected inside the called subprogram? Why can't you assign to the copy like other variables?

Answer: Because of compatibility issues. VB treats these entities as objects, and there is apparently no distinction between ByVal and ByRef for an object in VB -- both are equivalent to ByRef. In order to maintain upward compatibility with VB, BasicX implements ByVal as though it prohibits assigning to the object's properties.

Portability issues

This section addresses questions that arise if you want to port a Basic Express program to a Visual Basic compiler on a PC. Here we discuss language-related issues only.

Issues relating to differences between operating systems are not covered here. For example, VB programs commonly presuppose a windowing environment that includes a monitor, keyboard and mouse. None of those devices are necessarily present in an embedded system like BasicX, which means the usual built-in VB controls for command buttons, sliders and other windows-related objects are not covered.

Other operating system dependencies, such as serial I/O, are in the same category.

Checking for stack overflow

BasicX: Does not check for stack overflow.

VB: Checks for stack overflow.

Automatic variable initialization

BasicX: Does not automatically initialize variables.

VB: Automatically initializes variables. Numeric variables, for example, are initialized to zero.

Binary constants

BasicX: Includes system-defined binary constants, such as bx11110000.

VB: Binary constants are not built in, but they can be simulated by declaring the following symbolic constants in module-level code:

```
Public Const bx00000000 As Byte = &H00
Public Const bx00000001 As Byte = &H01
Public Const bx00000010 As Byte = &H02
  [...]
Public Const bx11111111 As Byte = &HFF
```

VB_Name attribute statements

BasicX: Ignores VB_Name attribute statements.

VB: Uses the VB_Name attribute to define the module name in which the attribute appears. Note that attribute statements are managed automatically by VB -- the statements are not normally visible in the source code when you use the VB environment for editing.

String arguments -- fixed length vs. variable length

BasicX: When a string variable is passed as an argument to a subprogram, it retains its identity as a fixed or variable length string. That is, fixed length strings stay fixed, and variable length strings stay variable.

VB: Converts all string arguments to variable length within the called subprogram. Only after return does a fixed length string recover its fixed length attribute.

Bounded strings

BasicX: Allows Mid assignment statements, such as Mid(S, 1, 2) = *expression*, where S is a bounded string.

VB: Does not allow Mid assignment statements for bounded strings. Mid functions are allowed, however.

UnsignedInteger and UnsignedLong types

BasicX: These types are provided by the system.

VB: These types are not supplied by the system, but they can be implemented as classes. Also, VB does not supply integer types that exactly match the numerical ranges for UnsignedInteger (0 to 65 535) or UnsignedLong (0 to about 4 billion). In most cases VB's 32-bit Long type is about the closest you can get for these objects, although VB's "Decimal" or "Currency" types could be used for UnsignedLong if you don't need to use bitwise Boolean operations on the type.

Mod operator

BasicX: Returns a positive or zero value regardless of the signs of the operands. For example:

    `(A Mod B)` is treated as equivalent to `(Abs(A) Mod Abs(B))`

VB: May return a negative result depending on the signs of the operands.

Scientific notation and data type

BasicX: If a numeric literal includes scientific notation, if neither a decimal point nor type declaration character is included, BasicX treats the number as a discrete type.

VB: Treats a numeric literal as a floating point type if scientific notation is used, regardless of whether a decimal point is present.

General error checking

If you want to port a program from BasicX to VB, differences in execution are generally minimized by selecting the following VB compiler options:

- Remove Array Bounds Checks

- Remove Integer Overflow Checks

- Remove Floating Point Error Checks

To get at these options in VB, go to the Project menu, choose Properties. Click on the Compile tab, then push the Advanced Optimizations button. The selections should appear.