

Optimize string handling in VB6 - Part I

Aivosto

String handling in Visual Basic is slow if done the wrong way. You can add significant performance to string operations by following some easy rules.

[Part I](#) | [Part II](#) | [Part III](#)

Faster strings with VB6

Visual Basic 6.0 offers a large selection of useful string handling functions such as [Left](#), [Mid](#), [Right](#), [Len](#), [Asc](#) and [InStr](#). They offer a powerful way operate with strings. Unfortunately, many of the string functions are not optimized for speed. This is why VB6 apps may run slower than necessary.

You can overcome many of the speed limitations by clever coding. This article shows a number of good tricks to add speed to string-intensive applications. The tricks use pure VB6 code. No extra run-time files or API calls are necessary.

In this article:

- [Why are VB6 strings so slow?](#)
- [Optimize the empty string](#)
- [No variants please](#)
- [Replace or not?](#)
- [Use constants](#)
- [Store strings in .res files](#)
- [Comparing strings](#)
- [String parameters](#)
- [Finding the bottlenecks](#)
- See also: [Part II](#), [Part III](#)

Related articles

[How not to optimize in Visual Basic](#)
[Optimize loops](#)
[Restructuring Visual Basic code](#)
[Save memory](#)
[Toolbox for project manager](#)
[VB InStr](#)
[VB tips: Optimize for memory and speed](#)

VB6 functions and operators in this article: [Asc](#), [AscW](#), [Chr\\$](#), [ChrW\\$](#), [Error\\$](#), [Format\\$](#), [Hex\\$](#), [InStr](#), [InStrB](#), [LCase\\$](#), [Left\\$](#), [Len](#), [LenB](#), [LTrim\\$](#), [Mid\\$](#), [Oct\\$](#), [Replace](#), [Right\\$](#), [RTrim\\$](#), [Space\\$](#), [Str\\$](#), [StrComp](#), [String\\$](#), [Trim\\$](#), [UCase\\$](#), [ByRef](#), [ByVal](#), [Like](#), [\\$](#).



PA This sign appears where [Project Analyzer](#) detects unoptimized coding. Project Analyzer is a VB code analysis tool that finds unoptimal functions and replaces them with better ones.

Who should read this article?

These tips are based on Visual Basic 6.0 and variable-length strings. They're most useful with string-intensive programs that read, parse or manipulate large amounts of text. The performance gains from using these techniques are significant if you're executing the calls thousands or hundreds of thousands of times. If you're just occasionally writing and reading a few strings outside of loops, these tips won't help you much. While the tips work best for VB6, some of them are generic in that they also apply to earlier and later versions of VB.

Why are VB6 strings so slow?

Perhaps the biggest bottleneck is that VB makes copies of the string data when doing some of the operations. Even when you're just reading strings (and not planning to make any modifications), you can easily end up making a large number of copies. The copying costs you time if string processing is an intensive part of your program. Another reason is that some of the widely used functions are implemented in a non-straightforward way. They may be doing more work than what is required for your task. Fortunately, you can often replace an advanced functions with a simpler and faster alternative.

Optimize the empty string

Does the `""` expression appear often in your code? Beware! So many CPU cycles are wasted for such a string! Testing and assigning empty strings is an easy place for optimization.

Checking for empty string

PA It's often necessary to test for an empty string. The usual ways are these:

```
If Text$ = "" Then  
If Text$ <> "" Then
```

However, VB executes the following equivalent statements much faster.

```
If LenB(Text$) = 0 Then  
If LenB(Text$) <> 0 Then
```

The replacement is essentially risk-free. Your code executes the same as before, only faster.

VB's implementation of `LenB` is fast. `LenB` is the byte equivalent of `Len`. `Len` is actually implemented as `LenB\2`. That makes `LenB` is faster than `Len`, so you should use it where possible. VB3 and VB.NET don't have the `LenB` alternative, in these languages you should use `Len`.

Note that we use the `<>` operator, not `>`. While `<>` simply tests for inequality, `>` tests more. As `Len/LenB` never return a negative number, we can safely use this test.

Assigning an empty string to a variable

PA This is the usual way to clear a string variable.

```
Text$ = ""
```

What a waste! First of all, the string `""` takes 6 bytes of RAM each time you use it. Consider the alternative:

```
Text$ = vbNullString
```

So what is this? `vbNullString` is a special VB constant that denotes a null string. The `""` literal is an empty string. There's an important difference. An empty string is a real string. A null string is not. It is just a zero. If you know the C language, `vbNullString` is the equivalent of `NULL`.

For most purposes, `vbNullString` is equivalent to `""` in VB. The only practical difference is that `vbNullString` is faster to assign and process and it takes less memory.

If you call some non-VB API or component, test the calls with `vbNullString` before distributing your application. The function you're calling might not check for a `NULL` string, in which case it might crash. Non-VB functions should

check for `NULL` before processing a string parameter. With bad luck, the particular function you're calling does not do that. In this case, use `""`. Usually APIs do support `vbNullString` and they can even perform better with it!

No variants please

It's a simple thing but often overlooked. All variables, parameters and functions should have a defined data type. If the data is a string, then the data type should be defined as string. If you don't give a data type, you're using a `Variant`. The `Variant` data type has its uses but not in string processing. A variant means performance loss in most cases.

PA So add those `Option Explicit` statements now and `Dim` all variables with a decent data type. Review your functions and ensure that they define a return data type.

Dollars that make your program run faster

PA The following functions are unoptimal if you're using them on strings:

```
Left(), Mid(), Right(), Chr(), ChrW(),  
UCase(), LCase(), LTrim(), RTrim(), Trim(),  
Space(), String(), Format(), Hex(), Oct(),  
Str(), Error
```

These are the dreaded `Variant` functions. They take a `Variant`, they return a `Variant`. These functions are OK to use if you really are processing `Variants`. This is the case in database programming, where your input may contain Null values.

So what's all that `Variant` stuff in string processing? It's fat. Forget about them if you're dealing with strings. Use the string versions instead. Just add a dollar and your program runs faster:

```
Left$(), Mid$(), Right$(), Chr$(), ChrW$(),  
UCase$(), LCase$(), LTrim$(), RTrim$(), Trim$(),  
Space$(), String$(), Format$(), Hex$(), Oct$(),  
Str$(), Error$
```

Dollar variables are no good

How about the dollar sign with variables? In the names of your own functions? Does it help?

No. The dollar sign only helps with the above VB functions. In this article we've also used the `$` sign to denote a string variable such as `Text$`. It's used for the sake of clarity. It's not an optimization tip!

PA We don't recommend the `$` sign for string variables. In real code, you should define your variables (and functions) with a real datatype, such as this: `Dim Text As String`. The dollar sign as a type indicator is obsolete.

Replace or not?

The following tip might be obvious, but it wasn't to us. It makes no sense to call `Replace` if you're not likely to replace anything. `Replace` runs slowly. `Replace` always creates a copy of the input string, even if no replacement occurs, and making the copy is slow. If a replacement is unlikely, verify first (with `InStr` or `InStrB`, for example) that there is something you need to replace.

```
If InStr(Text$, ToBeReplaced$) <> 0 Then  
    Text$ = Replace(Text$, ToBeReplaced$, "xyz")
```

End If

If a replacement is likely or certain to occur, there is no need to call `InStr`. It just adds an extra burden.

It's not necessary to add \$ in the call to `Replace`. This is an exception to the \$ rule. Adding the dollar makes no harm, but it makes no difference either. `Replace` and `Replace$` are the same function.

Use constants

Built-in string constants

PA Instead of calling `Chr$()` or `ChrW$()` on the following numeric values, use the predefined string constants. They will save you from the function call.

0	<code>vbNullChar</code>
8	<code>vbBack</code>
9	<code>vbTab</code>
10	<code>vbLf</code>
11	<code>vbVerticalTab</code>
12	<code>vbFormFeed</code>
13	<code>vbCr</code>
13 & 10	<code>vbCrLf</code>
13 & 10	<code>vbNewline</code>
34	<code>""</code>

For some reason, `vbNewline` is a little bit faster than `vbCrLf`.

The last example ("") is not actually a constant but an escape sequence. You can use "" anywhere in a string to represent a quotation mark. The alternative is `Chr(34)`, which was required in some early BASIC versions where the "" syntax didn't exist.

You can also define other other character values to avoid repeated calls to `Chr$()` or `ChrW$()`. If the character value is in the ASCII range 0 - 31, you need to define them as variables and assign the correct character value before use.

```
Dim BEL As String
BEL = ChrW$(7) ' The BEL character, or ^G
```

For other characters you can simply use a constant.

```
Const Percentage = "%"
```

Unnecessary Asc/AscW

PA It's obvious, but calling `Asc` or `AscW` on a string constant makes no sense. The value returned is a constant. It never changes. Instead of `Asc("A")`, use the value 65. Better yet, define a constant such as:

```
Const ascA = 65
```

Use the constant instead of 65 for more legibility. As it happens, VB.NET compiles `Asc("A")` better, but since we're in VB6, we need to define this constant.

Your own string constants

If the same string exists in more than one location in your project, it will also exist in several locations in the executable file, as far as VB6 is concerned (VB.NET joins duplicated strings during compilation).


You can optimize by defining your strings as constants and referencing the constant where you need the string value. This way you save space as each constant gets stored only once. Besides, if you ever consider localizing your program, you have a useful list of string constants to give to the translator.

There is a nasty exception. It doesn't save any space to define constants by other constants.

```
Const MSG1 = "Hello, "  
Const MSG2 = "world!"  
Const MSG3 = MSG1 & MSG2
```

In this case you will actually have the same text twice in the executable. All of `MSG1`, `MSG2` and `MSG3` will get stored – not something you wanted! If you want to save space, concatenate `MSG1` & `MSG2` at run-time. For speed, store it in a variable for reuse.

Also notice that the above applies to string constants only. Numeric constants are also computed and stored in the executable, but string constants are more likely to demand more space: 6 bytes overhead + 2 bytes per character.

PA [String literal analysis](#)  is a Project Analyzer feature that reports duplicate strings. Follow the link to read more about the elimination of unnecessary string literals.

Store strings in .res files

When compiling to an executable file, VB stores (most) string literals in Unicode, requiring 2 bytes per character. If you want to store your strings 1 byte per character, use resource files instead. This might reduce your executable size considerably if the amount of string data is large.

Note that you need to store the strings as a "custom resource" (binary format), not in the regular resource string table (Unicode). Press the Add Custom Resource button in the VB6 Resource Editor to add a text file as a custom resource.

Resource files are also handy for storing very long strings, multiline strings and strings that may be subject to localization.

Bug alert: If you store strings as a custom resource, make sure the strings consist of plain ASCII characters (0 – 127). Alternatively, make sure all the users use the same codepage as you. Otherwise the text may look different in a different locale. As an example, instead of the letter Ä a Greek user can see the letter Δ. The default way of storing strings as Unicode avoids this problem.

Comparing strings

Comparing strings against each other may take longer than you expected. Here are a few tricks.

Comparing the leftmost character

Here are two unoptimized ways to branch on the first character in a string.

```
' Case 1  
If Left$(Text$, 1) = "A" Then  
  
' Case 2
```

```

Select Case Left$(Text$, 1)
    Case "A"
    Case "B"
End Select

```

Rather than calling `Left$()`, we can call `AscW()` to determine the first letter of a string. The following examples are faster:

```

' Case 1
If LenB(Text$) <> 0 Then
    If AscW(Text$) = 65 Then
        ' AscW("A")=65

' Case 2
If LenB(Text$) <> 0 Then
    Select Case AscW(Text$)
        Case 65 ' A
        Case 66 ' B
    End Select

```

Calling `AscW()` is faster than first calling `Left$()`, then comparing the result to another string. There's a caveat, however. `AscW()` on an empty or null string is a run-time error. That's why you must first test with `LenB()` to rule out that possibility. You can leave out the call to `LenB()` only if you're certain that the string contains at least one character.

The `Select Case` structure offers an additional bonus. Having single numbers in the `Case` conditions is less time-intensive than repeatedly comparing against a string.

Comparing a character in the middle of a string

Similar to the above trick, this is the way to check for a character in the middle of a string.

```

If AscW(Mid$(Text$, index, 1)) = 65 Then

```

Note that index must be less than or equal to `Len(Text$)`. Otherwise you get a run-time error.

Note: If `Mid$` may return a long string, the third parameter to `Mid$(, , 1)` is essential for optimization. Without the parameter `Mid$` can spend a lot of time making an unnecessarily long copy of `Text$`. [Part III](#) of this article goes deeper into this issue.

Compare in binary

PA Whenever you can, use binary comparison. This is VB's default. Text comparison is much slower. These statements slow your application down:

```

Option Compare Text
StrComp(, , vbTextCompare)
InStr(, , vbTextCompare)

```

If you need a case-insensitive `StrComp()`, use `LCase$` to do it, especially if it's enough on one parameter only:

```

' Slower
StrComp(Text1$, "abc", vbTextCompare)

```

```
' Faster
StrComp(LCase$(Text1$), "abc", vbBinaryCompare)
```

In the following case, the two calls to `LCase$` remove the performance gain you got above:

```
StrComp(LCase$(Text1$), LCase$(Text2$), vbBinaryCompare)
```

Bear in mind that `StrComp(,vbTextCompare)` is more than just a case-insensitive comparison. It's actually built for sorting, not comparing for equality. In many cases, such a locale-dependent textual comparison is an overkill and can even lead to subtle errors. [More about StrComp](#)

Check for existence with InStrB

`InStr` is a nice function to find a string inside another one. Normally you use the plain `InStr` function, the wide-character version. [More about InStr](#)

There is an optimization with the byte version, `InStrB`. If you are just going to check whether a string exists inside the other but don't care about the location, you can use the following code:

```
If InStrB(Text$, SearchFor$) <> 0 Then
```

Since you only compare the return value against zero, you don't need to worry about conversions between byte-based indices and character indices. This is not the whole story, however. You need to be aware of the following catches:

- `InStrB` works completely on byte-based index values. The return value, as well as the start index parameter (the first numeric parameter, not present in the above call) are both in bytes, not in characters. One character is 2 bytes. Use the equation `byteindex = (characterindex * 2) - 1` to convert indices. If there is a match at character 3, the byte index is 5.
- `InStrB` is a byte data function and it's dangerous to use it on character input. If the strings may contain character values outside the range 1 – 255, be careful. Chances are `InStrB` is not good for you. As `InStrB` does a byte-wise search, it can return matches between characters:

```
' Bytes 34 12 78 56 hex
Text$ = ChrW$(&H1234) & ChrW$(&H5678)
' Bytes 12 78
SearchFor$ = ChrW$(&H7812)
```

In this case, `InStrB` returns 2, which is the start of the byte sequence 12 78 but doesn't match any of the input characters. This is probably not what you want when working with strings. Note that even if your strings are plain ASCII, the null character can still pose a problem. Example: `InStrB("A" & vbNullChar, vbNullChar)` returns 2, not 3 as one might expect.

- There are extra complications with the `vbTextCompare` parameter. `vbBinaryCompare` is simpler to understand.

What does this mean? Use `InStrB` to optimize only when you fully understand how it works.

Like

The `Like` operator is not particularly fast. Consider alternatives. We don't have a generic rule to follow here. You need to measure the performance differences between your alternatives. Here is one rule though. It applies if you're looking for a certain string inside another one.

Instead of `Like`:

```
If Text$ Like "*abc*" Then
```

use `InStr`:

```
If InStr(Text$, "abc") <> 0 Then
```

You may also use `InStrB` if you know what you're doing.

String parameters

Procedure string parameters differ from numeric parameters in that with strings, the chosen parameter passing convention makes a real performance difference.

Pass strings ByRef

How should you define procedure parameters for calls from within the same project?

`ByVal` is slow for string parameters. `ByVal` makes a copy of the string on every call. The good side is that a `ByVal` parameter is safe to modify: the modifications aren't passed back to the callers.

`ByRef` is faster because the string doesn't get copied. The drawback is that you have to be careful. If your intention is not to return a value in the `ByRef` parameter back to the caller, you may not accidentally write to this parameter.

Use ByRef instead of function return value

There's also an optimization trick for returning a string value. Returning a string as the function return value is the normal practice. However, returning a string in a `ByRef` parameter is faster.

The `ByRef` trick for return values applies to both functions and `Property Get`'s. Here's the usual (and slower) way:

```
' Slow:
Property Get Name() As String
    Name = m_sName
End Property
```

```
' Slow:
Function Name() As String
    Name = m_sName
End Function
```

This way is faster if you have to make a large number of calls:

```
' Fast:
Sub GetName(ByRef Name_out As String)
    Name_out = m_sName
End Sub
```

It's often considered bad programming style to return values in parameters. Normally procedures should not cause side-effects by modifying their `ByRef` parameters. However, if you want speed, you sometimes have to reject accepted programming practices to win a few CPU cycles. Thus, the optimization objective might justify the loss of style. You can use `ByRef`, but you should indicate why you're using it. For example, you can mark all output parameters with the word out, or write a comment saying `ByRef` is used for speed.

Stick to ByVal for out-of-process calls

There is one case where [ByRef](#) is slower than [ByVal](#). This happens when passing [ByRef](#) to an out-of-process server. The variable has to be marshalled twice, once going into the method and once returning. The implication is to use [ByVal](#) for your public server interfaces.

Finding the bottlenecks

Most of the performance gains by string optimization may actually be due to a limited number of changes in certain key locations in your code. What are these locations and how can you find them?

A critical location is executed thousands or hundreds of thousands of times. It may be inside a loop or a recursive algorithm. The location may consist of a handful of procedures or certain lines of code inside them.

It's not always possible to tell a bottleneck by just looking at the code searching for loops or recursion. A code profiler, such as [VB Watch](#), is useful for finding the critical locations. It logs the execution times as you run your program. When ready, it tells you which procedures or lines were executed the highest number of times, and which ones took the longest time to execute. These places are the best candidates for manual optimization work. For other parts of the code you can rely on a more automatic optimization method, such as letting [Project Analyzer](#) run [auto-fix](#) on your code by routinely replacing ineffective calls with better ones.

Advanced string optimization techniques

[Part II](#) introduces you to fast and slow VB functions, Unicode API calls and robust handling of huge strings.

Part I | [Part II](#) | [Part III](#)

PA This sign appears where [Project Analyzer](#) detects unoptimized coding. Project Analyzer is a VB code analysis tool that finds unoptimal functions and replaces them with better ones.

Related articles

[How not to optimize in Visual Basic](#)

[Optimize loops](#)

[Restructuring Visual Basic code](#)

[Save memory](#)

[Toolbox for project manager](#)

[VB InStr](#)

[VB tips: Optimize for memory and speed](#)

P.S. Flowchart your code? Try [Visustin](#).

Optimize string handling in Visual Basic 6.0

URN:NBN:fi-fe20061526

©Aivosto Oy - www.aivosto.com

vbshop@aivosto.com