Write Faster Apps with VB Pointers

Click & Retrieve Source CODE!

BY MATTHEW CURLAND AND FRANCESCO BALENA

Write amazingly fast routines using a hidden capability of VB: real pointers to memory.

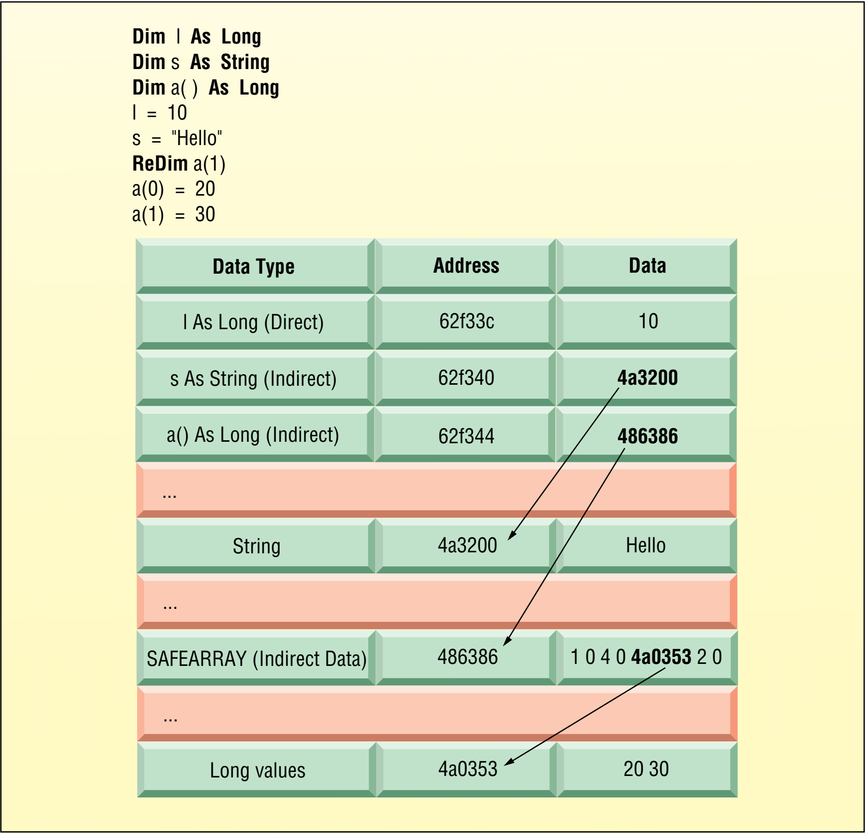
It's a common misconception that Visual Basic doesn't have pointers to memory. Although VB doesn't allow you to declare pointer variables or do pointer arithmetic, you can find the pointers inside some VB data types just under the surface. In fact, it is indeed possible to access a portion of memory as if it were a regular VB array. This technique opens up a number of profitable opportunities to VB programmers. You'll discover that you can write VB applications that are as fast as C/C++ programs, and solve problems that have always been thought to be out of the reach of VB programmers.

If you have ever programmed in a low-level language, chances are you already have some familiarity with memory pointers. It is nearly impossible to write a nontrivial C program without using pointers, and on the average, every other line in an assembly listing includes an explicit reference to a memory address. A few higher-level RAD tools, such as Borland Delphi, offer native access to memory pointers, although the pointer manipulation support is not as extensive as in C and C++. With Visual Basic, you can't use pointer variables.

Here's how memory pointers can be useful: imagine you need to read and modify all the individual pixels of a bitmap. These pixels are stored somewhere in memory; in fact, you can get their addresses easily. Unfortunately, if you program in pure VB, there isn't much you can do with this address because you have no built-in way to de-reference the pointer and access the actual data. This restriction makes intrinsic VB code safe from crashes, but puts artificial limits on memory usage and available algorithms.

Francesco Balena is editor in chief of Visual Basic Journal, *the Italian licensee of* VBPJ, *and cofounder of Software Design, a software firm specializing in VB and VC++ add-ons as well as training and consulting. He is a coauthor of* Platinum Edition Using Visual Basic 5 *(Que). Contact Francesco at fbalena@infomedia.it.*

Matthew Curland is a developer on the Visual Basic team at Microsoft. He is a coauthor of Object Programming with Visual Basic 4 *(Microsoft Press). To see a sample of his VB programming work (and get a really cool object browser), download a demo version of the Object Navigator tool from the Demos & Updates page at www.microsoft.com/mspress. Reach him at MattCur@Microsoft.com.*



```
Dim l As Long
Dim s As String
Dim a( ) As Long
l = 10
s = "Hello"
ReDim a(1)
a(0) = 20
a(1) = 30
```

Data Type	Address	Data
l As Long (Direct)	62f33c	10
s As String (Indirect)	62f340	**4a3200**
a() As Long (Indirect)	62f344	**486386**
...		
String	4a3200	Hello
...		
SAFEARRAY (Indirect Data)	486386	1 0 4 0 **4a0353** 2 0
...		
Long values	4a0353	20 30

FIGURE 1 ***These Variables Store Pointers.*** *VB string, array, and object variables are indirect. They store pointers to data instead of the data itself. Strings use single indirection and arrays use double indirection to access data.*

VB5 VB4 32-bit

```
Type SAFEARRAYBOUND
   cElements As Long
   lLbound As Long
End Type

  Type SAFEARRAY1D
      cDims As Integer
      fFeatures As Integer
      cbElements As Long
      cLocks As Long
      pvData As Long
      Bounds(0 To 0) As SAFEARRAYBOUND
  End Type

  Type SAFEARRAY2D
      cDims As Integer
      fFeatures As Integer
      cbElements As Long
      cLocks As Long
      pvData As Long
      Bounds(0 To 1) As SAFEARRAYBOUND
  End Type

  Type SAFEARRAY60D
      cDims As Integer
      fFeatures As Integer
      cbElements As Long
      cLocks As Long
      pvData As Long
      Bounds(0 To 59) As SAFEARRAYBOUND
End Type

Declare Function VarPtrArray Lib "msvbvm50.dll" _
  Alias "VarPtr" (Ptr() As Any) As Long
Declare Sub CopyMemory Lib "kernel32" Alias _
  "RtlMoveMemory" (pDst As Any, pSrc As Any, _
  ByVal ByteLen As Long)

Function ArrayMemory(ByVal arrPtr As Long) As Long
  ' this procedure should be passed the address
  ' of a SafeArray structure, as in
  '     bytes = ArrayMemory(VarPtrArray(myArray))
  Dim sa As SAFEARRAY60D, saPtr As Long
  Dim i As Long, numEls As Long, cDims As Integer

  ' retrieve the SafeArray structure
  CopyMemory saPtr, ByVal arrPtr, 4
  ' retrieve the dimension count
  CopyMemory cDims, ByVal saPtr, Len(cDims)
  CopyMemory sa, ByVal saPtr, _
    Len(sa) - 8 * (60 - cDims)
  ' evaluate the number of elements
  numEls = 1
  For i = 0 To sa.cDims - 1
    numEls = numEls * sa.Bounds(i).cElements
  Next
  ' evaluate total amount of memory
  ' including the SA structure itself
  ArrayMemory = numEls * sa.cbElements + _
    (16 + sa.cDims * 8)
End Function
```

LISTING 1 ***Show Your Byte Waste.*** *When passed the address of the descriptor of an array (any type, any number of dimensions), this routine returns the total amount of memory taken by the array, including the array descriptor itself. The declarations at the top are used by all the other listings in this article.*

A brave VB programmer might successfully move all the pixels to a VB array or matrix for processing, and then put them back in their original memory address. If you are dealing with an 800-by-600 256-color image, that means allocating about half a megabyte of local data and copying the contents of the memory block back and forth from the memory for the actual bitmap to your local copy. These crippling inefficiencies could be avoided if you could access the data where it really is—right in the bitmap.

You can make your apps run faster in VB by looking for ways to use pointers inside VB data types. Data held in VB variables can be broken down into two categories: direct data and indirect data. For direct data—such as Integer, Long, Currency, or user-defined types—a VB variable contains the memory location of the data. For indirect data, the variable contains the memory location of a pointer to the data. Indirect types include objects, strings, and arrays of any type (see Figure 1).

We will show you how to use the underlying pointer data types in VB to directly manipulate data at any memory location. We'll explore the SAFEARRAY structure that underlies all VB arrays, demonstrating how to make a VB array variable point to arbitrary data.

Use extreme caution when applying these techniques. VB assumes that it allocated any data it has a pointer to, but the pointer data demonstrated here is borrowed, so VB can't free it without crashing. All clean-up code shown is absolutely necessary.

UNLOCKING SAFEARRAYS

A VB array variable holds a pointer to a SAFEARRAY structure, which looks like this:

```
Type SAFEARRAY
   cDims As Integer
   fFeatures As Integer
   cbElements As Long
   cLocks As Long
   pvData As Long
End Type
Type SAFEARRAYBOUND
   cElements As Long
   lLbound As Long
End Type
```

A SAFEARRAY structure is always allocated in memory with a fixed number of SAFEARRAYBOUND structures immediately following it. The number of bounds corresponds to the number of dimensions in the array, and can't be changed without destroying and re-creating the SAFEARRAY. A one-dimensional VB array looks like this:

```
Type SAFEARRAY1D
   cDims As Integer
   fFeatures As Integer
   cbElements As Long
   cLocks As Long
   pvData As Long
   Bounds(0 To 0) As SAFEARRAYBOUND
End Type
```

Let's review the fields in the SAFEARRAY type. cDims is the number of dimensions in the array and always corresponds to the number of bounds. fFeatures is used to indicate various array properties, such as whether the data is fixed-size or points to strings, objects, or variants that must be freed before the array data is destroyed. Because we don't allow VB to free any of the SAFEARRAYs we construct, we don't use fFeatures in this article.

cbElements is the size, in bytes, of a single element in the array. For example, a Long array has an element size of four bytes. As with fFeatures, we don't discuss the intricacies of cLocks. pvData is a pointer to the actual data; it is the pivotal field for this article. cElements is the number of elements in an array bound and lLbound is the number corresponding to the lower bound of the array.

After you set up the SAFEARRAY structure, the next step in direct pointer manipulation is making a VB array variable point to your data. In other words, given the variables *psa() As Byte* and *sa As SAFEARRAY1D*, how do you make psa(0) correspond to the first byte of data in the memory location pointed to by sa.pvData? In C, the variables look like *SAFEARRAY* psa* and *SAFEARRAY sa*, and the assignment code looks like *psa = &sa* (we'll actually use **(&psa) = &sa)*. In VB, because direct pointer assignment isn't supported natively, we have to call on some helper functions.

The first helper function is VarPtr, which is built into VB5, although hidden. VarPtr returns the address of any non-array VB variable, so VarPtr(sa) is equivalent to &sa in C. However, the VarPtr function doesn't accept an array variable, so you need to add an aliased declaration to the VarPtr entry point in the VB runtime DLL that accepts an array of almost any type. The only type that doesn't work is () As String because VB does ANSI/Unicode conversion on String arrays, but not UDTs containing strings. An easy workaround is to use an array of a dummy type that contains a single String element:

```
Declare Function VarPtrArray Lib _
   "msvbvm50.dll" Alias "VarPtr" _
   (Ptr() As Any) As Long
```

Now, VarPtrArray(psa) gives you a pointer to the psa variable (&psa). You can use the CopyMemory API call to make the final step:

```
Declare Sub CopyMemory Lib "kernel32" _
   Alias "RtlMoveMemory" _
   (pDst As Any, pSrc As Any, _
   ByVal ByteLen As Long)
'*(&psa) = &sa
CopyMemory ByVal VarPtrArray(psa), VarPtr(sa), 4
```

Before you make the CopyMemory call, you must make sure that psa doesn't currently point to another array. Use the Erase statement if you're not sure. You must declare your array variable as variable size—that is, don't specify dimensions in the Dim statement—or you will leak memory. After you're done using the array and before psa goes out of scope, you must clear the psa variable with a second CopyMemory call to prevent VB from freeing the data. Be sure to use 0& (a Long), not 0 (an Integer) as the second parameter:

```
CopyMemory ByVal VarPtrArray(psa), 0&, 4
```

You can also use VarPtrArray and CopyMemory to get information such as the number of dimensions and size of each element from an existing SAFEARRAY. You can't get this data through built-in VB functions. If you don't know how to read a SAFEARRAY structure, you need an error-trapped loop containing a UBound(arr, n) statement to find the dimensions, and the LenB function to make a guess as to the size of an array element. LenB is unreliable because it doesn't detect packing between array elements:

```
'psa is any non-empty VB array
Dim sa As SAFEARRAY
Dim saPtr As Long
CopyMemory saPtr, ByVal VarPtrArray(psa), 4
CopyMemory sa, ByVal saPtr, Len(sa)
Debug.Print sa.cDims, sa.cbElements
```

By simply reading the SAFEARRAY structure associated with an array, you can learn interesting things. For instance, you can write a generic function that returns the amount of memory used by any array, regardless of its type and number of dimensions (see Listing 1).

Writing such a routine is tricky because you don't know how many dimensions the array has. Therefore, you don't know how many SAFEARRAYBOUND records you must allocate and read. Even if you know that VB arrays support a maximum of 60 dimensions, you cannot simply allocate a fixed array of 60 items and fill them with a single CopyMemory call, because you can cause a general protection fault (GPF) merely by reading outside the block of memory allocated to the SAFEARRAY structure. Instead, you need two distinct CopyMemory calls: the first one to get cDims and the second one to read the correct number of SAFEARRAYBOUND items. Also, in order to work around VB's inability to accept As Any arguments in VB procedures, you must pass this routine the address of an array descriptor, as in:

```
bytes = ArrayMemory(VarPtrArray(myArr))
```

A quick word on VB4: all the techniques we discussed so far work in VB4/32 as well as VB5, except that VarPtr isn't built into VB4. But you can declare VarPtr in your VB4 code directly. As long as you don't pass the VB4-declared VarPtr function any strings or structures containing strings, the two calls work the same.

The techniques up to this point are also usable in VB4/16. However, 16-bit applications use handled memory with SAFEARRAYs, so any code using pvData is unlikely to work the same. In fact, the 16-bit SAFEARRAY structure is slightly different. SAFEARRAYs aren't used in VB3. The VB4/32 declares look like this:

VB5 VB4 32-bit

```
Sub ArrayAdd(arr() As Long, ByVal increment As Long)
  ' add a given constant value to all the items
  ' of a N-dimension array of Longs
  Dim sa As SAFEARRAY1D, sa2 As SAFEARRAY60D
  Dim saPtr As Long, i As Long, numEls As Long
  Dim cDims As Integer

  ' retrieve information on matrix dimensions
  ' and number of elements
  CopyMemory saPtr, ByVal VarPtrArray(arr), 4
  CopyMemory cDims, ByVal saPtr, Len(cDims)
  CopyMemory sa2, ByVal saPtr, _
    Len(sa2) - 8 * (60 - cDims)
  ' evaluate the number of elements
  numEls = 1
  For i = 0 To sa2.cDims - 1
    numEls = numEls * sa2.Bounds(i).cElements
  Next

  ' create a temp array and retrieve its SA structure
  ReDim temp(0) As Long
  CopyMemory saPtr, ByVal VarPtrArray(temp), 4
  CopyMemory sa, ByVal saPtr, Len(sa)
  ' modify the number of elements and pointers to data
  sa.pvData = sa2.pvData
  sa.Bounds(0).cElements = numEls
  ' have the descriptor point to our local SA
  CopyMemory ByVal VarPtrArray(temp), VarPtr(sa), 4
  ' add the increment to all elements
  ' (temp is zero-based)
  For i = 0 To numEls - 1
    temp(i) = temp(i) + increment
  Next
  ' reset temp() descriptor to the original value
  CopyMemory ByVal VarPtrArray(temp), saPtr, 4
End Sub
```

LISTING 2 ***Process Matrices for Speed.*** *Because multidimensional matrices are accessed more slowly than monodimensional arrays, it is often much faster to deal with a large matrix as if it were a regular array. This routine adds a constant to a matrix of Long (any number of dimensions): it is twice as fast as using two nested For…Next loops, and about four times faster when processing three-dimensional matrices.*

VB5 VB4 32-bit

```
Private m_fInit As Boolean
Private Const cstrHexPairs As String = _
  "000102...DFEFF"
Private m_psaHexPairs(255) As Long

Public Function BinToHex(bytes() As Byte) As String
Dim saForString As SAFEARRAY1D
Dim psaForString() As Long
Dim saForBytes As SAFEARRAY1D
Dim saPtrForBytes As Long
Dim lCount As Long
  'Get the safearray data for the passed in array.
  'This gives us a fast mechanism to look at the
  'dims, lower bound, and element count of the array.
  CopyMemory saPtrForBytes, ByVal VarPtrArray(bytes), 4
  CopyMemory saForBytes, ByVal saPtrForBytes, _
  Len(saForBytes)
  With saForBytes
    If .cDims <> 1 Then Err.Raise 5
    'Normalize the lower bound to 0,
    'save calculations in loop
    If .Bounds(0).lLbound Then
      CopyMemory ByVal UnsignedAdd( _
        saPtrForBytes, 20), 0&, 4
    End If
    'Make sure array of hex values is initialized
    If Not m_fInit Then
      CopyMemory m_psaHexPairs(0), ByVal _
        StrPtr(cstrHexPairs), LenB(cstrHexPairs)
      m_fInit = True
    End If

    'Allocate returned string array directly
    'into name of function
    BinToHex = String$(.Bounds(0).cElements * 2, 0)
    'Set up safearray pointing to string
    'we just allocated
    With saForString
      .cbElements = 4
      .cDims = 1
      .pvData = StrPtr(BinToHex)
    End With
    saForString.Bounds(0).cElements = _
      .Bounds(0).cElements
    CopyMemory ByVal VarPtrArray(psaForString), _
      VarPtr(saForString), 4

    'Do the actual work to copy the hex bytes across.
    lCount = .Bounds(0).cElements
    Do While lCount
      lCount = lCount - 1
      psaForString(lCount) = _
        m_psaHexPairs(bytes(lCount))
    Loop

    'NULL out array pointer
    CopyMemory ByVal VarPtrArray(psaForString), 0&, 4
    'Restore old lbound
    If .Bounds(0).lLbound Then
      CopyMemory ByVal UnsignedAdd( _
        saPtrForBytes, 20), .Bounds(0).lLbound, 4
    End If
  End With
End Function

Private Function UnsignedAdd _
  (Start As Long, Incr As Long) As Long
  ' only works with positive increments
  If Start And &H80000000 Then 'Start < 0
    UnsignedAdd = Start + Incr
  ElseIf (Start Or &H80000000) < -Incr Then
    UnsignedAdd = Start + Incr
  Else
    UnsignedAdd = (Start + &H80000000) + _
      (Incr + &H80000000)
  End If
End Function
```

LISTING 3 ***Manipulate Numeric Strings.*** *Process strings as if they were numeric arrays to avoid the string routines in the Visual Basic runtime.*

```
Declare Function VarPtr Lib _
   "vb40032.dll" (Ptr As Any) As Long
Declare Function VarPtrArray Lib _
   "vb40032.dll" Alias "VarPtr" _
   (Ptr() As Any) As Long
```

CHANGING ARRAY ATTRIBUTES

Now that you know how to read a SAFEARRAY structure, you can have some real fun by modifying it. You can modify the attributes of a VB array using three techniques. The first one is to act on the individual fields of the original SAFEARRAY structure in memory. For instance, you can modify the first index to the array—the value returned by the LBound function—without redimensioning it:

```
Dim sa As SAFEARRAY1D
Dim saPtr As Long
'read the SafeArray structure
CopyMemory saPtr, ByVal _
   VarPtrArray(psa), 4
CopyMemory sa, ByVal saPtr, Len(sa)
'modify and copy it back to memory
sa.Bounds(0).lLBound = 1
CopyMemory ByVal saPtr, sa, Len(sa)
'at this point psa is one-based
```

Changing the LBound limit of an array in this way is safe, and you don't even need to undo changes before VB frees the array and its associated memory. You can't modify any other field in the SAFEARRAY structure without putting everything back exactly as it was before VB releases the array. If you fail to correctly restore the array data, VB releases a block of memory it didn't allocate and you will experience a steady stream of GPFs when your arrays go out of scope.

The second technique for modifying array attributes is to have the array variable temporarily point to another SAFEARRAY structure—either an existing one that belongs to another array or one allocated by yourself. In this case, you must have the descriptor point back to the original SAFEARRAY structure before VB destroys the array.

For example, you can modify the cElements and pvData item of a temporary array and use it to access the memory owned by a multidimensional matrix (see Listing 2 and Figure 2). You can look at the matrix as if it were a one-dimensional array, which is often much faster. When using this trick, remember that VB matrices are stored one column after the other.

This second approach works flawlessly, but it requires that you dimension an array—temp() in this case—only to assign a SAFEARRAY to its descriptor. You can keep memory overhead to a minimum using one-item arrays, but you can't avoid a call to allocation/release routines, which are relatively slow. To write amazingly fast routines, avoid this overhead.

With the third technique for handling SAFEARRAY structures, you declare an array of proper type, but do not dimension it at all. Instead, you prepare a suitable SAFEARRAY structure and force the address of this structure into the array descriptor. You can rewrite the ArrayAdd routine like this:

```
Dim temp() As Long
Dim sa As SAFEARRAY1D
With sa
   .cDims = 1
   .cbElements = 4
   .pvData = sa2.pvData
   .Bounds(0).cElements = numEls
End With
```

```
CopyMemory ByVal VarPtrArray(temp), _
   VarPtr(sa), 4
```

Before the array goes out of scope, we must force the descriptor back to null, because temp() must appear to VB as an undimensioned array:

```
CopyMemory ByVal VarPtrArray(temp), _
   0&, 4
```

PROCESS STRINGS FAST

We'll use a binary-to-hexadecimal conversion routine as a sample (BinToHex). Writing BinToHex with native string manipulation routines such as Hex$ and the Mid$ statement is easy to code, but it's unacceptably slow. To make a screaming routine (ours does more than 5 MB of data per second on a P6/200), you need to have pure numeric, rather than string, calculations. To do hex conversion, you need to visit each byte, so regardless of the algorithm you use, you have a loop that takes the majority of your processing cycles. You can minimize the code in this loop using SAFEARRAYs.

The first step is to initialize your static data. A byte has 256 possible values, which correspond to 256 two-character hex pairs. Put these in a module-level string constant to avoid calculating them at run time.

Next, copy this data into a fixed-size Long array the first time you call BinToHex. The choice to use a fixed-size array is a trade-off. It requires some extra memory—it's possible to share the memory used by the string—but eliminates the need for clean-up code. Use the built-in StrPtr function (similar to VarPtr) to retrieve a pointer to the constant string data. Unfortunately, you can't declare StrPtr in VB4/32 without creating a type library.

```
Private Const cstrHexPairs As String = _
   "000102…FDFEFF"
Private m_psaHexPairs(255) As Long
Private m_fInit As Boolean
'...
If Not m_fInit Then
   CopyMemory m_psaHexPairs(0), _
      ByVal StrPtr(cstrHexPairs),
      LenB(cstrHexPairs)
   m_fInit = True
End If
```

The Long value in each element of m_psaHexPairs is the numeric representation of the corresponding hex pair, so you have numeric input data. Now set up a corresponding Long array as a destination for the hex pair data. Your function and variable declarations look like this:

```
Public Function BinToHex(bytes() As _
   Byte) As String
Dim saForString As SAFEARRAY1D
Dim psaForString() As Long
Dim saForBytes As SAFEARRAY1D
Dim saPtrForBytes As Long
Dim lCount As Long
```

Your optimized main loop looks like this:

```
lCount = saBytes.Bounds(0).cElements
Do While lCount
   lCount = lCount - 1
   psaForString(lCount) = _
      m_psaHexPairs(bytes(lCount))
Loop
```

Perform two more preparatory steps before using the loop. Step one is to allocate the string buffer and link it to the psaForString array, which lets you write directly to the memory you're returning. The second step, which is less obvious, is to make sure that the input array has a lower bound of zero, which eliminates an extra shift calculation for each pass of the loop. As a prerequisite to these steps, retrieve information from the input *bytes* array using the code shown earlier. saForBytes holds a copy of the SAFEARRAY descriptor, and saPtrForBytes is a pointer to the original structure. Allocate

VB5 VB4 32-bit

```
Type BITMAP
   bmType As Long
   bmWidth As Long
   bmHeight As Long
   bmWidthBytes As Long
   bmPlanes As Integer
   bmBitsPixel As Integer
   bmBits As Long
End Type

Declare Function GetObjectAPI Lib _
   "gdi32" Alias "GetObjectA" (ByVal _
   hObject As Long, ByVal nCount As _
   Long, lpObject As Any) As Long

 Sub SwapColors(pictbox As PictureBox, ByVal color1 _
   As Integer, ByVal color2 As Integer)
 ' these are used to address the pixel using matrices
 Dim pict() As Byte


 Dim sa As SAFEARRAY2D, bmp As BITMAP
 Dim r As Integer, c As Integer, value As Byte

 ' get bitmap info
 GetObjectAPI pictbox.Picture, Len(bmp), bmp
 ' exit if not a supported bitmap
 If bmp.bmPlanes <> 1 Or bmp.bmBitsPixel <> 8 Then
   MsgBox " 256-color bitmaps only", vbCritical
   Exit Sub
 End If

 ' have the local matrix point to bitmap pixels
With sa
   .cbElements = 1
   .cDims = 2
   .Bounds(0).lLbound = 0
   .Bounds(0).cElements = bmp.bmHeight
   .Bounds(1).lLbound = 0
   .Bounds(1).cElements = _
      bmp.bmWidthBytes
   .pvData = bmp.bmBits
End With
 CopyMemory ByVal VarPtrArray(pict), VarPtr(sa), 4

 ' swap colors - note that col/row order is inverted
 ' because VB arrays are stored in column-wise order
 For c = 0 To UBound(pict, 1)
   For r = 0 To UBound(pict, 2)
     value = pict(c, r)
     If value = color1 Then
       pict(c, r) = color2
     ElseIf value = color2 Then
       pict(c, r) = color1
     End If
   Next
 Next

 ' clear the temporary array descriptor
 ' without destroying the local temporary array
 CopyMemory ByVal VarPtrArray(pict), 0&, 4
 ' inform VB that something has changed
 pictbox.Refresh

 End Sub
```

LISTING 4 ***Process Images Cheaply.*** *Use this routine to swap any pair of colors in a 256-color bitmap held in a picture box control, and as a starting point for building more interesting graphic effects.*

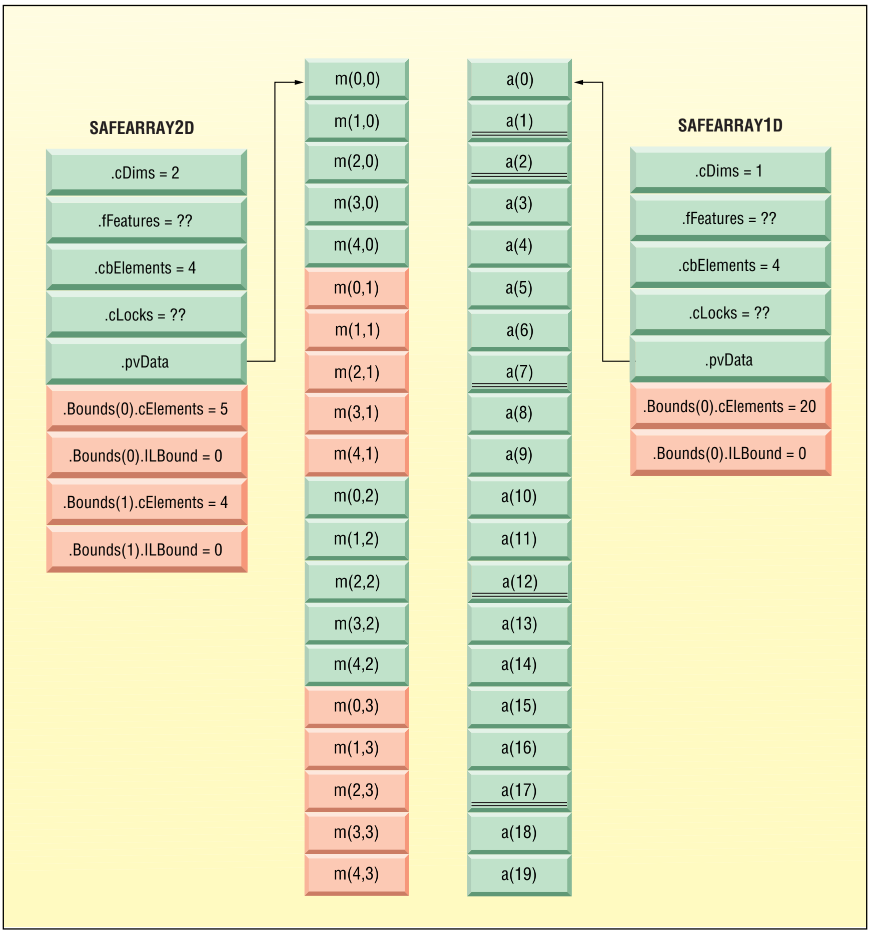


FIGURE 2 ***Take a Second Look at Your Matrices.*** *If you can handle the SAFEARRAY structure, you can access any portion of memory as if it were a native VB array. Moreover, you can look at an existing multidimensional matrix as if it were a linear array, which is often more efficient. However, remember that VB matrices are stored one column after the other.*

the string directly into the name of the function instead of a temporary string, to avoid an extremely expensive extra string assignment at the end of the routine:

```
BinToHex = String$(saBytes.cElements _
   * 2, 0)
With saForString
   .cbElements = 4
   .cDims = 1
   .pvData = StrPtr(BinToHex)
   .cElements = saForBytes.cElements
End With
CopyMemory ByVal VarPtrArray( _
   psaForString), VarPtr(saForString), 4
```

To normalize the byte array, use CopyMemory to write directly to cElements, which is offset 20 bytes from saPtrForBytes. Because VB doesn't have an unsigned long data type, which is necessary for pointer arithmetic, use the provided UnsignedAdd function to treat signed long data as a pointer. Return the lower bound of the input array to its original value before leaving BinToHex:

```
If saForBytes.Bounds(0).lLbound Then
   CopyMemory ByVal _
   UnsignedAdd(saPtrForBytes, 20), _
   0&, 4
End If
```

With error checking at the beginning of the function, and code to remove the borrowed pointer from psaForString and reset the lower bound of the input array, you're done with your function (see Listing 3). The amount of setup code in this function is excessive for manipulating small amounts of data, but is well worth the effort for large amounts of data when the loop is processed thousands of times. Also, the single line that allocates the string buffer often takes more time than the rest of the setup code, depending on the size of the string. This procedure runs best in native code with bounds checking and integer overflow checking turned off.

MANIPULATE MEMORY FOR IMAGE PROCESSING

Serious image processing in VB might seem like an oxymoron, but it's possible with the techniques you've learned so far. After all, an image is nothing more than a bidimensional array of pixels in memory, and you now know how to manipulate this memory efficiently.

Getting the address of the pixels in a bitmap is easy. Given a bitmap in a picture box control, you can pass its handle to the GetObject API—returned by the Picture property—and fill a BITMAP structure with the relevant information:

```
Type BITMAP
   bmType As Long
   bmWidth As Long
   bmHeight As Long
   bmWidthBytes As Long
   bmPlanes As Integer
   bmBitsPixel As Integer
   bmBits As Long
End Type
Declare Function GetObjectAPI Lib _
   "gdi32" Alias "GetObjectA" _
   (ByVal hObject As Long, _
   ByVal nCount As Long, _
   lpObject As Any) As Long
…
Dim bmp As BITMAP
GetObjectAPI Picture1.Picture, _
   Len(bmp), bmp
```

The fields of interest are bmWidth (the width of the bitmap in pixels), bmHeight (its height), bmWidthBytes (the width of pixel rows, where each row is aligned on a word boundary), and bmBits (the memory address of actual pixels). In our example, we only deal with 256-color bitmaps with one color plane (bmPlanes field) and eight bits per pixel (bmBitsPixel field).

The 256-color bitmaps are easy to deal with because each pixel corresponds to a byte, which holds the index of the color in the bitmap's palette. You can build a suitable SAFEARRAY structure and point an uninitialized array variable at the descriptor:

```
Dim pict() As Byte, sa As SAFEARRAY2D
With sa
   .cbElements = 1
   .cDims = 2
   .Bounds(0).lLbound = 0
   .Bounds(0).cElements = bmp.bmHeight
   .Bounds(1).lLbound = 0
   .Bounds(1).cElements = bmp.bmWidthBytes
   .pvData = bmp.bmBits
End With
CopyMemory ByVal VarPtrArray(pict), _
   VarPtr(sa), 4
```

Finally, you can access individual pixels as if they were elements of the pict() matrix, with one important difference: VB bidimensional arrays are stored column-wise, while bitmaps are stored row-wise. Therefore, we need to swap the usual row/column indices:

```
'color of pixel at row 20, column 40
value = pict(40, 20)
```

Take a look at a complete routine that swaps a pair of colors in a bitmap, effectively changing all pixels of color1 into color2, and vice versa (see Listing 4). The routine is not aware of the actual color of the pixels it is dealing with; it just sees numbers in the range 0–255. For more advanced image processing techniques—color reduction, for instance—you must use this value as an index into the bitmap's palette to retrieve the real RGB value. We won't cover such added intricacies here, because our primary concern is memory pointers, not image processing.

Pointers that directly read and write memory locations offer a huge number of possible applications, besides the ones described in this article. However, it helps to remember that even though you're using the data structure that underlies all VB arrays, using these pointers cheats VB into doing something it was not designed to do. If you follow our directions closely, you will end up with robust and super-fast applications, but don't forget to save your work often while you're testing and debugging. ☒

Code Online

You can find all the code published in this issue of **VBPJ** ***on The Development Exchange (DevX) at http://www.windx.com. All the listings and associated files essential to the articles are available for free to Registered members of DevX, in one ZIP file. This ZIP file is also posted in the Magazine Library of the*** **VBPJ** ***Forum on CompuServe. DevX Premier Club members ($20 for six months) can get each article's listings in a separate file, as well as additional code and utilities for selected articles, plus archives of all code ever published in*** **VBPJ** ***and*** **Microsoft Interactive Developer** ***magazines.***

Write Faster Apps with VB Pointers

Locator+ Codes

Listings ZIP file (free Registered Level): VBPJ1097

✪ Listings for this article plus a routine to rotate any bitmap, a super-fast ParseString routine, and complete programs for benchmarking string conversion and rotating bitmap routines (subscriber Premier Level): MC1097P