Use the Windows API for Text Handling Inside VFP
Richard Aman
Ricka@Bellsouth.net

The Windows clipboard can hold many different data formats and corresponding data handles, all representing the same data, but in as many different formats as an application is able to supply. For example, a pie chart might be held in the clipboard as both a metafile picture and a bitmap. An application pasting the pie chart would have to choose which representation best matched its requirements. In this article, Richard Aman discusses how to use the Windows API for cutting and pasting data instead of the internal Visual FoxPro cut and paste functions.

Visual FoxPro 3.0, like most Windows products, provides internal methods for copying to and pasting from the Windows clipboard. These internal functions move the selected text between the applications and the clipboard. When the user presses the appropriate hot-key combination (usually Ctrl-C to copy and Ctrl-V to paste), the internal functions move the selected objects to and from the clipboard automatically and under Windows control.

So why bypass the internal FoxPro text editing functions and use custom functions instead? I can think of three reasons: for security, data integrity, and interapplication communication. Let's discuss the circumstances where these concerns might be valid. However, the example form I provide deals only with the issue of data security.

Data security -- prevent sensitive data from being copied
Before users are allowed to copy or paste text it may be necessary to check the text that the users have selected to see if it contains any sensitive information. Ideally, the users aren't prevented from working with the text using the normal text editing functions while in the source application, but are prevented from moving the text to another application. For example, if users are in a memo field of a form that contains sensitive information, they should be able to use all normal editing functions, except they should not be allowed to move the data to a word processor or the Windows Notepad.

One way to accomplish this would be to encode the data at the time the users copy it and decode the data when it is pasted. If users paste the data in another application, the data would be useless without decoding.

Here are a couple of simple encoding/decoding routines that are used in the sample form included in the accompanying Download file.

```
*-------------------------------------------------------
*-- Encode()
*-- This function is a simple text encoder. All it does is
*-- scan the given text string and increment each ASCII code
*-- by one in the string, then return the encoded string.
*--
*PARAMETERS tcText
*-- tcText is the text to be translated
*-- define variables
PRIVATE lcChar
PRIVATE lnLoop1
PRIVATE lcText
*-- init variables
lcChar = ''
lnLoop1 = 0
lcText = ''
*-- begin main process
FOR lnLoop1 = 1 TO LEN( tcText )
  lcChar = MIDD( tcText, lnLoop1, 1 ) + 1
  lcText = lcChar + lcText
NEXT
RETURN lcText
END
```

Data integrity – validate the data before pasting
By bypassing the native FoxPro editing functions, checks can be performed on the data before it is pasted. For example, if users are in one field on a form, copy data to the clipboard, and then enter another field to paste the data, the data can be validated before the users paste the data in the new edit field rather than waiting for the users to exit the field and only then performing validation. The sooner users are made aware of incorrect data, the more likely they will still be focused on that particular data.

Inter-application communication using OLE automation
By using the Windows API clipboard functions for text editing, custom functions can be expanded to allow data to be passed between applications via the clipboard and OLE automation or e-mail. For example, users can select data from a field on a form and send the data to an Excel spreadsheet or a Word document for further processing. Or a custom e-mail message can be automatically generated and sent along its way.

The two main editing functions
Copying and pasting are the two main functions used when working with text data. To bypass Visual FoxPro 3.0's internal editing functions and use custom functions instead, the application has to gain direct access to the clipboard. I accomplish this with the help of several functions from the Windows API and two wrapper functions I wrote to encapsulate the functionality of the Windows API.

The implementation involves trapping the native Ctrl-C and Ctrl-V keypresses in a form and replacing them with these custom functions. This is done in the Load event of the form and is explained later in this article. First, however, I'll discuss these custom functions -- CopyText(), a replacement for Ctrl-C, and PasteText(), a replacement for Ctrl-V.

CopyText()

*******************************************************************************
* FUNCTION CopyText
*******************************************************************************
* Author.............: Richard L. Aman
* E-Mail.............: 73700.141@compuserve.com
* Project............: Clipboard Usage Within VFP 3.0
* Created.........: 05/27/96  16:31:45  
* Copyright.......: (c) 1996, Aman Data Systems  
* Description.....: This program demonstrates the steps  
* Calling Samples.: =CopyText( tcSelectedText )  
* Parameter List..: tcSelectedText - the text to copy  
* Change list.....: 
#include <windows.h>

LPARAMETERS tcSelectedText

LOCAL lcSelectedText  
LOCAL GHND  
LOCAL CF_TEXT  
LOCAL MAXSIZE  
LOCAL hGlobalMemory  
LOCAL lpGlobalMemory  
LOCAL lpClipMemory  
LOCAL hClipMemory  

*-- declare Windows API functions for this module  
DECLARE INTEGER GlobalUnlock IN kernel32 INTEGER  
DECLARE INTEGER GlobalLock IN kernel32 INTEGER  
DECLARE INTEGER GlobalAlloc IN kernel32 INTEGER, INTEGER  
DECLARE INTEGER lstrcpy IN kernel32 INTEGER, STRING  
DECLARE INTEGER OpenClipboard IN user32 INTEGER  
DECLARE INTEGER EmptyClipboard IN user32  
DECLARE INTEGER CloseClipboard IN user32  
DECLARE INTEGER SetClipboardData IN user32 INTEGER, INTEGER  

* Check to see if there's anything to copy to the clipboard  
IF LEN( tcSelectedText ) = 0  
WAIT WINDOW "Nothing to copy right now"  
RETURN  
ENDIF  

*-- NOTE: the following is only to be used  
*-- if it is necessary to encode the text  
*-- before it is copied to the clipboard  
*-- encode the string before copying to the clipboard  
lcSelectedText = thisForm.Encode( TRIM( tcSelectedText ) )  
IF LEN( lcSelectedText ) = 0  
WAIT WINDOW "The encoder returned a zero-length string. ;  
Copy aborted"  
RETURN  
ENDIF  

* note: GHND = 66 which initializes a block of moveable  
* global memory to 0's  
hGlobalMemory = GlobalAlloc( GHND, ;  
LEN( lcSelectedText ) + 1 )  
IF ISNULL( hGlobalMemory )  
WAIT WINDOW "Could not allocate the memory block. ;  
Copy aborted"  
RETURN  
ENDIF  

* lock the block to get a far pointer to the global memory  
lpGlobalMemory = GlobalLock( hGlobalMemory )  
IF ISNULL( lpGlobalMemory )  
WAIT WINDOW "Could not lock the memory block. ;  
Copy aborted"  
RETURN  
ENDIF  

*-- copy the string to the global memory  
lpClipMemory = lstrcpy( lpGlobalMemory, lcSelectedText )  
IF ISNULL( lpClipMemory )  
WAIT WINDOW "Could not copy the text to the clipboard. ;  
Copy aborted"  
RETURN  
ENDIF
Let's assume that the application needs to transfer a character string to the clipboard. First, allocate a moveable global memory block of LEN(lcSelectedText) size. Include room for a terminating NULL:

The value of hGlobalMemory() will be NULL if the block couldn't be allocated. If the allocation is successful, lock the block to get a far pointer to it:

Copy the character string into the global memory block. Since the string is NULL-terminated, the application can use the Windows string-copy function lstrcpy():

The terminating NULL doesn't need to be added because the GHND flag for GlobalAlloc() zeroes out the entire memory block during allocation. Unlock the block:

Now the application has a global memory handle that references a memory block containing the NULL-terminated text. To get this into the clipboard, open the clipboard:

It's always a good idea to clear the clipboard to ensure that no data is left over from the previous use:

Copy the text to the clipboard by giving the clipboard the global memory handle using the CF_TEXT identifier:

When finished, close the clipboard to allow other applications to use the clipboard:

All done. The text has been transferred to the clipboard and is accessible from this and other applications.

Some notes concerning copying to the clipboard
Call OpenClipboard() and CloseClipboard() within the same procedure. Don't leave the clipboard open when the procedure exits. Don't let control transfer to another program while the clipboard is open.

Don't give the clipboard a locked memory handle.

After the call to SetClipboardData(), don't continue to use the global memory block. It no longer belongs to the application, and
the handle should be treated as invalid. If the application needs to continue to access the data, it should make another copy of it or read it from the clipboard (as described in the next section). The application can also continue to reference the global memory block between the SetClipboardData() call and the CloseClipboard() call, but must use the global handle that is returned from SetClipboardData(). Be sure to unlock this handle before the call to CloseClipboard():

**PasteText()**

*******************************************************************************
* FUNCTION PasteText
*******************************************************************************
* Author ..........: Richard L. Aman
* E-Mail ..........: 73700.141@compuserve.com
* Project ..........: Clipboard Usage Within VFP 3.0
* Created ..........: 05/27/96  16:31:45
* Copyright ......: (c) 1996, Aman Data Systems
*) Description ....: This program demonstrates the steps
*) .................: necessary to paste data from the clip
*) .................: board using Windows API function calls
* Calling Samples .: =PasteText()
* Parameter List .:
*** define variables
LOCAL CF_TEXT
LOCAL MAXSIZE
LOCAL hClipMemory
LOCAL lpClipMemory
LOCAL lcMyString
LOCAL lnRetVal
*** init variables
CF_TEXT = 1
MAXSIZE = 4096
lpClipMemory = 0
hClipMemory = 0
lcMyString = ''
lnRetVal = 0
*** declare Windows API functions for this module
DECLARE INTEGER OpenClipboard IN user32 INTEGER
DECLARE INTEGER CloseClipboard IN user32
DECLARE INTEGER GetClipboardData IN user32 INTEGER
DECLARE INTEGER GlobalAlloc IN kernel32 INTEGER, INTEGER
DECLARE INTEGER GlobalLock IN kernel32 INTEGER
DECLARE INTEGER GlobalUnlock IN kernel32 INTEGER
DECLARE INTEGER lstrcpy IN kernel32 STRING @, INTEGER
DECLARE INTEGER GlobalSize IN kernel32 INTEGER, INTEGER
DECLARE INTEGER IsClipboardFormatAvailable IN user32 INTEGER
* check to see if there is any text available for pasting
IF IsClipboardFormatAvailable( CF_TEXT ) <> 1
    WAIT WINDOW "No text available in the ;
    clipboard right now."
RETURN
ENDIF
*** try to open the clipboard
IF OpenClipboard(0) = 0
    WAIT WINDOW "Could not open the Clipboard. ;
    Another application may have it open"
RETURN
ENDIF
*** obtain the handle to the global memory
*** block that is referencing the text
hClipMemory = GetClipboardData( CF_TEXT )
IF ISNULL( hClipMemory )
    WAIT WINDOW "Could not allocate memory"
ELSE
*** lock clipboard memory so we can reference
*** the actual data string
lpClipMemory = GlobalLock( hClipMemory )
*** copy string from clipboard to local variable
IF NOT ISNULL( lpClipMemory )
    lcMyString = SPACE( MAXSIZE )
    lnRetVal = lstrcpy( @lcMyString, lpClipMemory )
    lnRetVal = GlobalUnlock( hClipMemory )
*** peel off the null terminating character.
lcMyString = SUBSTR( lcMyString, 1, AT( CHR(0), ;

Getting text from the clipboard is only a little more complex than transferring text to the clipboard. First, the application must determine whether the clipboard does in fact contain data in the CF_TEXT format. One of the easiest methods is to use this call:

```c
bAvailable = IsClipboardFormatAvailable( CF_TEXT )
```

This function returns TRUE (nonzero) if the clipboard contains CF_TEXT data. IsClipboardFormatAvailable() is one of the few clipboard functions that can be used without first opening the clipboard. However, if the clipboard is opened later to get this text, the application should check again (using the same function or one of the other methods) to determine if the CF_TEXT data is still in the clipboard. To transfer the text out, first open the clipboard:

```c
OpenClipboard( 0 )
```

Obtain the handle to the global memory block referencing the text:

```c
hClipMemory = GetClipboardData( CF_TEXT )
```

This handle will be NULL if the clipboard doesn't contain data in the CF_TEXT format. This is another way to determine if the clipboard contains text. If GetClipboardData() returns NULL, close the clipboard without doing anything else.

The handle received from GetClipboardData() doesn't belong to the application -- it belongs to the clipboard. The handle is valid only between the GetClipboardData() and CloseClipboard() calls. The application can't free that handle or alter the data it references. If the application needs to have continued access to the data, it should make a copy of the memory block.

Lock the handle to the clipboard memory:

```c
lpClipMemory = GlobalLock( hClipMemory )
```

Because the character string is NULL-terminated, the data can be transferred using Windows lstrcpy() function:

```c
lstrcpy( @lcMyString, lpClipMemory )
```

Unlock the global memory handle:

```c
GlobalUnlock( hClipMemory )
```

Peel off the terminating NULL:

```c
lcMyString = SUBSTR( lcMyString, 1, AT( CHR(0), ; lcMyString ) - 1 )
```

Finally, close the clipboard:

```c
CloseClipboard()
```

Now the application has the clipboard text in the variable lcMyString for use within the application.

### A few other considerations for clipboard use

Here are a few other considerations when using the Windows API functions to access the clipboard from within a Visual FoxPro application:

Only one program can open the clipboard at any time. The purpose of the OpenClipboard() call is to prevent the clipboard contents from changing while a program is using the clipboard. OpenClipboard() returns a logical value indicating whether the clipboard was successfully opened. It won't be opened if another application failed to close it. During the early stages of programming for the clipboard, it's probably wise to check this value, but the check isn't crucial in a nonpreemptive...
multitasking environment. If every program politely opens and then closes the clipboard during a single procedure without giving control to other programs, then being unable to open the clipboard should never be a problem.

Another subtle problem to avoid involves message boxes: If a global memory segment can't be allocated, the system should normally display a message box. If this message box isn't system modal, however, users can switch to another application while the message box is displayed. Either the message box should be a system modal message box, or the clipboard should be closed before the message box is displayed.

Dialog boxes can be another source of trouble: If the clipboard is left open while displaying a dialog box, remember that the edit fields in a dialog box also use the clipboard for cutting and pasting text.

Also, if the application encodes and decodes data while copying and pasting, this will also prevent users from copying from another source and pasting into the application unless the decoding function is bypassed when pasting the text.

**Using the functions on a form**

Here's an example of how to use these functions on a form (this form is also included as part of the Download file):

1. Start Visual FoxPro 3.0 and create a new form.
2. Add four new methods to the form and name them as follows:
   - **CopyText** -- this will hold the code to call when Ctrl-C is pressed.
   - **PasteText** -- this will hold the code to call when Ctrl-V is pressed.
   - **Encode** -- this will hold the code to encode the clipboard text.
   - **Decode** -- this will hold the code to decode the clipboard text.
3. Copy the code for CopyText into the method.
4. Copy the code for PasteText into the method.
5. Copy the code for Encode into the method.
6. Copy the code for Decode into the method.
7. In the Load() method of the form, add the following two lines of code:
   ```foxpro
   SET SYSMENU OFF
   SET SYSMENU TO DEFAULT
   ```
   This disables Visual FoxPro's internal copy and paste methods and will allow the new CopyText() and PasteText() functions to be called when Ctrl-C and Ctrl-V are pressed.

In the form's UnLoad() method add the following line of code:

```foxpro
SET SYSMENU TO DEFAULT
```

This resets the system menu back to its default.

Add two edit boxes to the form and resize them to a comfortable width for working with lines of text. In the KeyPress() method of the first edit box, add the following three lines of code:

```foxpro
IF nKeyCode = 3 AND nShiftAltCtrl = 2
   thisForm.CopyText( this.SelText )
ENDIF
```

In the KeyPress() method of the second edit box, add the following four lines of code (the keyboard command is included to get rid of a spurious chr(14) that the API call wants to paste in along with the text):

```foxpro
IF nKeyCode = 22 AND nShiftAltCtrl = 2
   this.Value = thisForm.PasteText()
   keyboard " (backspace)"
ENDIF
```

Test out the clipboard functions:

1. Save and run the form.
2. Type some text in the first edit box.
3. Highlight the text in the first edit box.
4. Press Ctrl-C to copy the highlighted text to the clipboard.
5. Tab to the second edit box.
6. Press Ctrl-V to paste the text from the clipboard to the second edit box.
7. Start Notepad.
8. Press Ctrl-V to paste the text from the clipboard to the Notepad. (The text should be encoded.)

**Conclusion**
In the wonderful world of software development, there are usually as many ways to perform a function as there are developers. I’ve presented another way of copying and pasting text in Visual FoxPro 3.0 via the clipboard and the Windows API. By intercepting the text before it is copied to the clipboard and after it is copied from the clipboard, all sorts of operations can be performed on the text. Using the Windows API functions can open up a whole new world and give the developer more control over the application from within Windows. Every time I start digging around in the Windows API, I always seem to stumble upon some useful function or two that makes programming easier, more interesting, and more enjoyable. The documentation about the Windows API functions on the Developer’s Network Library and Technet CDs are an excellent source of information on the inner workings of the Windows API. I’ve come to appreciate what great tools the Windows API offer developers as we create Windows applications. The Windows operating system now does a lot of work for us that we used to have to do ourselves.