QCard32.DLL Documentation

Table of Contents

NOT ALL SMOKE AND MIRRORS

QCard32.DLL Subs and Functions

Once you have included the sub and function declarations in your Global Module, you can call QCARD32.DLL functions just as if you were calling any other function or sub in VB. The following sections list each function and sub and describes what it does. In this documentation, any parameter which begins with n, such as nCard, indicates that that value should be an integer. Any parameter which begins with b, such as bValue, indicates that that value should be a Boolean, TRUE or FALSE. CALLING THESE SUBS AND FUNCTIONS WITH VALUE TYPES OTHER THAN THOSE INDICATED IS A GOOD WAY TO LOCK UP YOUR SYSTEM! Always save your work before you run a routine to test it, especially the Block Dragging routines. If you pass these routines an improper value, you will be dumped out of VB and any of your unsaved work will be lost. Not a big deal, that's all part of the Windows programming experience!

Just remember to save your work first.

BOOL InitializeDeck(hWnd)

Only call this function once in your application. Did you catch that? I said "ONLY CALL THIS FUNCTION ONCE IN YOUR APPLICATION!!". One call to this function is all that's required for QCARD32.DLL to operate; any more calls than that, and you will get screwy things happening in your game. This sets up all the card pictures and values within the DLL. You can make this call in your Form.Load event, passing it the handle of the window in which you will be working. This function returns TRUE if it is successful and FALSE if it fails. You should always check the return value of this function and act accordingly if it fails. For example:

```
nReturnValue = InitializeDeck(Form1.hWnd)
If nReturnValue = FALSE Then
MsgBox "Unable to Initialize QCard32.DLL"
' bail out now!
End
End If
```

Generally, the InitializeDeck function will only return a FALSE value if the DLL is unable to load up its card bitmaps. In the 16-bit version of Qcard.DLL, only one application could use the DLL at one time. In the 32-bit environment, this restriction is removed. When your application ends, QCARD32.DLL is released and unloaded by Windows. You may find that if your application dies and ends prematurely when you are designing and running it, due to a General Protection Fault on your part, Windows might not properly unload the DLL. You will have to restart Windows again to clear the DLL out of memory before you can continue debugging your game. Under normal conditions, the DLL will be unloaded automatically by Windows.

SetCurrentBack(nIndex)

In addition to the DrawBack Sub, which draws one of six card back designs in your window, you can also use the regular card drawing and dealing subs with card numbers 105 through 109 to draw card backs. Cards numbered 105 through 109 act just like other cards, but their picture is a card back design rather than a card front design. This allows you to manipulate 5 facedown cards in the same way you can the other regular cards. The picture is the same for all five cards and is initially set at cardback design number 1. Call SetCurrentBack(nIndex), where (nIndex) is a number between 1 and 6, to change these cards to a different design. You may call SetCurrentBack(nIndex) any number of times to change card backs during your game, but remember to re-draw any previously dealt face-down cards to reflect the new choice.

You can also use cards 105 through 109 to display a pile of cards that dwindles as the user clicks on the pile, as in Windows Solitaire. To achieve this effect, first draw the "O" symbol on your form. Then deal card 105 directly on top of it. Then deal cards 106 through 108, each time offsetting their x and y by 2. This creates a nice 3-D stack effect. You will need to block all the cards except the top one (108 in this example). As an example, part of your MouseDown Event might look like this:

```
Dim Shared nTopCard As Integer
nTopCard = 108
nSourceCard = InitDrag(Form1.hWnd, x, y)
If nsourceCard = nTopCard Then
RemoveCard nTopCard
SetCardDisabled nTopCard, TRUE
AdjustCardBlocked nTopCard - 1, FALSE
nTopCard = nTopCard - 1
End If
AbortDrag
```

This partial code sample is just to get you started. To create a fully developed card pile, you will need to add much more functionality to the routine.

SetDefaultValues()

Use this sub to reset all card properties back to their default values. A good time to use this is right before setting up a fresh deal, so you can be sure all previous values are flushed out. It has no parameters. Call this routine as many times as you want during the course of your game.

SetCardStatus(nCard, bValue)

This Sub allows you to change any cards from their default setting of faceup to facedown. If you set a card's status to facedown, i.e., SetCardStatus (1, FALSE), it will be treated as facedown by the DLL when it is drawn or dragged. The image used for the facedown image will be that set by the SetCurrentBack(nIndex) Sub. Simply set the Status of any cards in your game which will be dealt facedown to FALSE, and then deal them as normal. They will be drawn facedown. To flip them faceup, set their Status to TRUE with SetCardStatus(nCard, TRUE), and use DrawCard to draw them faceup at their current location.

SetOffSet(nValue)

If you want to use QCard32.DLL for dragging blocks of cards in your game, you must deal cards in vertical columns as in Windows Solitaire. By default, cards in each column should be offset 16 pixels down from each other. If you wish to use a different vertical spacing in your game (other than 16 pixels), inform the DLL of the new spacing value with this Sub. nValue is the new value which will be used by the

DLL to carry out block dragging.

DrawCard (hWnd, nCard, nxLoc, nyLoc)

This is the quickest and easiest way to draw a card onto your window. Simply pass it your window handle, the number of the card you want drawn, and the x, y location you want the card to appear at. The DrawCard sub does not update any of the card's data members, such as its x or y location. If your application does not require any of this other information, this may be the only drawing sub you need to use. You cannot implement dragging operations if this is the only sub you use to draw your cards, however. Again, this does not update any of the card's data members. It just draws the card on the screen. It is fast and simple. (This makes it good for redrawing items for screen updates, as well).

DealCard (hWnd, nCard, nx, ny)

The DealCard sub does many important things over and above the DrawCard sub. It updates the card's X and Y properties to the location you deal the card. Most importantly, it grabs from the video display that portion of the screen your card will be covering over. That is, it keeps a copy of the image which lies behind that card. This is very crucial when you go to drag a card, because whatever used to be behind the card has to be replaced on the video display. If you are going to be doing any dragging, you must place your cards on the screen using the DealCard sub. If you try to drag a card that was originally drawn using the DrawCard sub alone, you will end up with a video mess.

Please note: the cards in QCARD32.DLL adapt to any background color your window may have. You can feel free to include an option for the user to change window colors in your game knowing that funny colored corners will not appear on the cards if the background color changes. One warning, however. As has been mentioned, each card carries with it a copy of the screen image which lies behind the card, for dragging purposes. If you deal the cards on a green background, and the user changes color to a red background, your card's background images will still reflect a green background. Not a pretty sight when he starts dragging! When changing screen colors in midstream, you should:

- Remove your cards from the screen
- Repaint the window to the new color
- Use the DealCard sub to replace your
active cards at their proper location

Doing this will ensure that their background images correspond to the present background color.

DrawSymbol (hWnd, nValue, nx, ny)

This sub draws the basic X, O and place holder symbols. It requires the hWnd of your form, a symbol value and an x and y position where you want the symbol drawn. Valid values are 1 for an X, 2 for an O, and 3 for the place-holder. These symbols feature a gray background rather than the usual black. They will show up on any background color, including very dark colors.

DrawBack (hWnd, nValue, nx, ny)

This sub draws one of the six included cardback designs at the location x, y. Valid values are 1 through 6 inclusive. Use Card numbers 105 to 109 if you want actual cards that show the current cardback as their image. The DrawBack Sub only draws a picture of the selected cardback on the screen.

RemoveCard (hWnd, nCard)

This sub removes the card from the window display assuming two things are true: First, the card must have been originally placed on the window using the DealCard sub. Second, the card must not be overlapped from above in any way. This sub actually repaints the card's background image where it used to be.

GetCardColor(nCard)

This function returns the color of the card specified. It returns 1 for a black card, 2 for a red card.

GetCardSuit(nCard)

This function returns the suit of the card specified. It returns 1 for Clubs, 2 for Diamonds, 3 for Hearts, 4 for Spades.

GetCardSuit(nCard)

This function returns the suit of the card specified. It returns 1 for Clubs, 2 for Diamonds, 3 for Hearts, 4 for Spades.

GetCardStatus(nCard) AND SetCardStatus(nCard, bStatus)

This Function and Sub pair can be used to get the current faceup/facedown status of a card, and also to change the current faceup/facedown status of a card. All cards originally have a Status of TRUE, or faceup. If you want some cards to be facedown, set their Status to FALSE with SetCardStatus(nCard, FALSE). Those cards will be handled as facedown by the DLL when they are subsequently drawn, dealt or dragged.

GetCardBlocked(nCard) AND AdjustCardBlocked(nCard, bValue)

These get and set the IsBlocked value of a card. You will have to pay particular attention to this property if you will be doing any dragging. Imagine Windows Solitaire. When the user presses the mouse button down over a card, the cursor may actually be over many cards in a pile. A technique is required which allows the application to determine which of those cards should actually be selected. QCARD32.DLL uses a method of putting a block on all cards covered over by another card. This is your responsibility to maintain.

When initializing a drag event, QCARD32.DLL first checks for any unblocked cards under the mouse cursor. If it finds one, it will return the number of that card. This initiates a Single drag operation. If it doesn't find one, it then determines if it is in the top 16 (or user defined OffSet) pixels of any other card, including blocked cards. If it is, it returns the number of that card. This initiates a Block drag operation. If you maintain your cards in proper blocked and unblocked fashion, you will have no trouble dragging single or groups of cards in the same way as Windows Solitaire. When creating a row or pile of cards, only the topmost card should have an IsBlocked value of FALSE. Remove a block by calling AdjustCardBlocked(nCard, FALSE). Block a card by calling AdjustCardBlocked(nCard, TRUE).

IsCardDisabled(nCard), SetCardDisabled(nCard, bValue)

You can set a card's Disabled property to TRUE so it can no longer be selected by the mouse for drag operations. Again, imagine Windows Solitaire. Even when cards are played to their top, final locations, they can be dragged down again and replaced on the lower piles. If you do not want "finished" cards to be replayed like that in your game, you can set their Disabled property to TRUE with SetCardDisabled(nCard, TRUE). Then the user will no longer be able to drag them back down into play.

GetCardX(nCard) AND GetCardY(nCard)

Use these functions to get the x and y location properties for a card. These are pixel coordinates based on 0, 0 in the top left corner of the window you are working in. These, along with SetCardX and SetCardY, are very useful for drawing and relocating cards around your window. For example, when the user drags a card over onto a new pile and lets it go, you will want to relocate it and snug it up below the previous card:

```
nDestCard = EndDrag Form1.hWnd, x, y
nNewX = GetCardX(nDestCard)

nNewY = GetCardY(nDestCard)
RemoveCard Form1.hWnd, nSourceCard
' using the default value of 16 for OffSet
DealCard Form1.hWnd, nSourceCard, nNewX, nNewY + 16
```

SetCardX(nCard) AND SetCardY(nCard) Subs

Use these two Subs to change the current X or Y setting of a card. Although this changes the value of X or Y, it does not move the card to the new location.

GetUsern(nCard) & SetUsern(nCard, nValue)

Use these subs and functions to get and set values of your choosing which you can associate with any of your cards. User1 is a Boolean TRUE and FALSE value. User2, User3 and User4 are all Integer types. You can use these any way your application requires. For example, you can call SetUser4 12, 1000 to set a User value of 1000 to card 12, and retrieve that value later by calling nMyValue = GetUser4(12).

These can come in handy in a variety of ways. In the demo program, they are used them to keep track of which array each card belongs to and its position in the array. This makes it easy to move cards from one pile (array) to another as the game executes.

InitDrag(hWnd, nx, ny)

Use this function in a MouseDown event to start a drag operation. The function searches through all cards to determine if the mouse cursor is over any card whose IsBlocked property is FALSE. If it finds one, it returns the number of that card. If it does not find one, it searches through all the cards in the deck to see

if the mouse lies in the top 16 (or User-Defined OffSet) pixels of any card, blocked or not. If it does, it returns the number of that card.

By checking the IsBlocked property of this returned card, you can tell if the user wants to carry out a single drag or a block drag. If InitDrag returns a value of 0, the mouse is not currently located over any card. The InitDrag function should always be followed by a corresponding AbortDrag, an EndDrag or an EndBlockDrag call. This is due to that fact that InitDrag "captures" all mouse input, and requires one of these corresponding calls to release the capture. The values nx and ny are the current mouse coordinates.

AbortDrag() Sub

This sub ends any drag operation started by an InitDrag call. AbortDrag releases the mouse which is captured by InitDrag. Abort drag takes no other action.

DoDrag(hWnd, nx, ny) Sub

Carries out the drag operation which was initiated by InitDrag call. DoDrag moves the current Source Card to its new location. Values nx and ny are the current mouse coordinates.

BlockDrag(hWnd, CardList(0), nNumCards, nx, ny)

BlockDrag carries out a block drag operation which was initiated by an InitDrag call. It requires a list of cards to be dragged in the form of an array. The array can be passed to the sub using the array's first element (0). The sub also requires the number of cards to be dragged as well as the current mouse coordinates.

EndDrag(hWnd, nx, ny)

EndDrag ends a single drag operation and returns the number of the Destination card (that is, the card it is being dropped on), if any. It searches the deck for any card which overlaps the Source Card and whose IsBlocked property is FALSE. If it finds one, it returns the number of that card. The function also releases the mouse which was captured by the InitDrag call.

EndBlockDrag(hWnd, CardList(0), nNumCards, nx, ny) Function

EndBlockDrag ends a block drag operation which was initiated by an InitDrag call. The function searches through the deck for any card which overlaps the Source Card and whose IsBlocked property is FALSE. If it finds one, it returns the number of that card. The function also releases the mouse which was captured by the InitDrag call. The function requires a list of the cards being dragged in the form of an array. The array can be passed to the function using the array's first element. The function also requires the number of cards being dragged and the current mouse position.

ReturnDrag(hWnd, nCard, nxLoc, nyLoc)

This sub drags the card nCard to the location nxLoc, nyLoc along a straight line from its current location. Return drag can be used for returning cards to their original location after an invalid drag operation. (That is, drag operations that your game determines to be invalid).

ReturnBlockDrag(hWnd, CardList(0), nNumCards, nxLoc, nyLoc)

This sub drags a block of cards to the location nxLoc, nyLoc along a straight line from their current location. It can be used to return a block of cards to their original location if your game determines the user has dragged them somewhere they shouldn't be.

DRAGGING

Although dragging is made easier using QCARD32.DLL, it is still a complex operation and one that will prove a little tough when you first try to implement it. Still, if your application is well organized and thought out, you will be able to include dragging operations with no problems.

A Simple Single Drag Example

When dragging cards, you will need to provide code for three events in relation to the Form you are working with. These are the MouseDown, MouseMove and MouseUp events. In the MouseDown event, you will initialize the drag operation. In the MouseMove event, you will carry out the drag operation. In the MouseUp event, you will end the drag operation. Of course, these events are happening all the time as your application is running, so you will need a switch to indicate whether a drag is in progress or not. For this purpose, create a Shared Integer variable in your General section which you can set to the Boolean values of TRUE and FALSE as your application runs:

```
Dim Shared bDragging
```

Initially, in your Form's Load procedure, this should be set to FALSE. In this simple example, begin by dealing a single card on your form:

```
DealCard Form1.hWnd, 1, 10, 10
```

This will deal the Ace of Clubs at location 10, 10 on your form. In response to the MouseDown event, we need to determine if the mouse is currently on the card or not. If it is, we can set the bDragging switch to TRUE. If not, we need to cancel the drag operation by calling the AbortDrag sub. The following code handles the MouseDown event:

```
Dim nSourceCard as Integer
nSourceCard = InitDrag(Form1.hWnd, x, y)
if nSourceCard = 0 Then
AbortDrag
Else
bDragging = TRUE
End If
```

The InitDrag function does several things. First, it takes the x, y mouse coordinates you pass it, and it looks through all the cards in the deck to see if any card lies underneath that location. If there is a card at that location and its IsBlocked and Disabled properties are both FALSE, it returns the number of that card. We can assign this value to the nSourceCard variable. In this case, this value will be 1, since that is the only card we have dealt on our form. If the InitDrag function cannot find a card at that mouse location, it will return a value of 0. In this case, we will need to abort the drag operation. Always follow up an InitDrag call with either an AbortDrag call or an EndDrag call. One of the things that InitDrag does is capture all mouse movements. You need to release the mouse capture by calling either AbortDrag or EndDrag, otherwise your application will effectively lock up your system since it is collecting all mouse information and directing it only to itself.

If no card is selected, we can abort the drag right here. If one is selected, we will release the mouse in the MouseUp event procedure. In the MouseMove event procedure, we need to test whether or not a drag is in progress. If it is, we will carry out the drag, if not, we don't need to do anything. Here is the MouseMove procedure:

```
If bDragging = TRUE Then
DoDrag Form1.hWnd, x, y
End If
```

The DoDrag sub moves the card to its new location and updates its X and Y properties accordingly. The card which moves is the one selected by the InitDrag function. If bDragging = FALSE, nothing happens.

In the MouseUp event procedure, we need to end the drag if one is in progress. Here is where we will call the Endrag function, thereby releasing the mouse capture:

```
Dim nDestCard As Integer
If bDragging = TRUE Then
nDestCard = EndDrag(Form1.hWnd, x, y)
bDragging = FALSE
End If
```

The EndDrag function does several things. Most importantly, as mentioned, it releases the mouse capture so mouse information can once again go to other applications other than this one. It also relocates the card to its new location and updates its X and Y properties accordingly. Finally, it checks to see if any other card lies beneath the card that has just been dragged and dropped. If there is a card that you have just covered over and that card's IsBlocked property is FALSE, the EndDrag function will return the number of that card. We assign that value here to the nDestCard variable. If there is no such card beneath the just dragged card, the function returns 0.

In this example, we declared the two variables nSourceCard and nDestCard as local to their respective Subs. In a real application, you would declare them as Shared or Global so you could carry out some comparison and testing on them. Since we now know the number of the Source card and the number of the Destination card, we could test whether or not this is a valid drag. In a game like Windows Solitaire, for example, where you can place a card of one less value on top of a card of the opposite color, some of the testing might look like this:

```
Dim nSourceColor As Integer
Dim nSourceValue As Integer
Dim nDestColor As Integer
Dim nDestValue As Integer
Dim bValidDrag As Integer
nSourceColor = GetCardColor(nSourceCard)
nDestColor = GetCardColor(nDestCard)
nSourceValue = GetCardValue(nSourceCard)
nDestValue = GetCardValue(nDestCard)
If nSourceColor nDestColor And nSourceValue = nDestValue - 1 Then
bValidDrag = TRUE

Else
bValidDrag = FALSE
End If
```

QCARD32.DLL provides a nice little routine for handling invalid drags. In your MouseDown event procedure, you can save the original location of the Source Card before dragging it. Declare two Shared or Global variables called OldX, and OldY, and assign them as follows in your MouseDown procedure:

```
OldX = GetCardX(nSourceCard)
OldY = GetCardY(nSourceCard)
```

Then, in your MouseUp event procedure, if you determine through comparison that this Source Card does not really belong on this Destination Card, then you can send it back to where it came from by calling:

```
ReturnDrag Form1.hWnd, nSourceCard, OldX, OldY
```

The ReturnDrag sub will automatically drag a card from its current location to the x, y location specified. It drags the card along a nice straight line. You may also find the ReturnDrag sub useful in other situations.

Avoiding Problems

To avoid problems, just think of the cards in your game as being real cards in a three dimensional sense. At any given time, some of the cards in your game will be standing alone in the open while others will be blocked in a pile with other cards. Problems arise when you allow the user to do a Single Drag operation on a card which is covered over by another card. This produces some unsightly video results. Instead, you must think in terms of "Last Card On, First Card Off". In doing this, you must dynamically maintain the IsBlocked status of all the cards in your game. Only free standing cards in the clear should have an IsBlocked status of FALSE. All other cards should have an IsBlocked status of TRUE.

Although you do not have to use arrays in your game, arrays representing piles of cards makes things much easier to maintain. As cards are dragged from one pile to another, you can update the IsBlocked properties for the affected cards in each array. For example, if dragging a single card from the bottom of one row of cards to the bottom of another row, you would unblock the last card which was freed up in the original row, block the last card in the destination row which is now covered by the new card, remove the card from the original row's array of members, and add the new card to the destination row's array of members. If you think the problem out carefully, you can arrange the data in your game so this process is handled automatically as your game executes. See the demo program for an example of one way to do this.