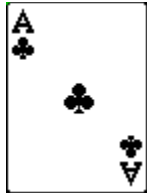


Contents



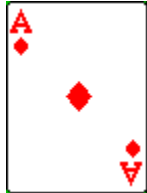
QCARDS

[Introduction](#)



USING QCARDS FOR FUN & PROFIT

[Cards up your sleeves](#)



Subroutines & Functions

[Not All Smoke And Mirrors](#)

[Initialization Functions](#)

[Card drawing subs](#)

[Information Subs and Functions](#)

[Dragging Functions](#)



Dragging it all out...

[DRAGGING](#)

[Dragging a Block of Cards](#)

Errata...

[ABOUT QCARDS.DLL](#)

Introduction

What is QCard.DLL?

QCARD.DLL is a dynamic link library that simplifies the creation of card games for the Windows 3.x environment. It grew out of the author's desire to write his own card games which were as good in quality as the Windows solitaire game. There were some other card DLL's available, but they proved to be somewhat slow, and the final result was not always that pretty.

QCARD.DLL is fully compatible with the Visual Basic environment as well as C. If you are working in VB you might encounter slightly slower execution speed but this is the nature of VB. **QCARD.DLL** gives you easy access to 40 card-game specific functions and procedures. Creating a Windows card game has never been easier

*I do this for fun, not for money! Therefore the **QCARD.DLL** is released to an unsuspecting world as FREEWARE and offers no ties or binds to it's author. The other benefit of releasing this as FREEWARE is being able to avoid the thirteen paragraphs of legal mumbo-jumbo that would normally go right here.*

CARDS UP YOUR SLEEVES

Writing Card Games

Using VB and QCard.DLL

QCard.DLL Card Properties

Writing Card Games

Writing a card game can be a frustrating process even when using tools like this. **QCARD.DLL** will not write your card game for you! You will need to do lots of work behind the scenes to ensure that you are passing **QCARD.DLL** correct information in the correct way. The process is eased by the amount of fore-thought you bring to it. Like any other programming project, the more time you spend designing and thinking about what you want to do, the easier it is once you actually sit down and start coding it. I am as guilty as the next person when it comes to just sitting down and banging out wonderfully creative code with no regard to where it is heading. I usually end up paying the price, which is involves lengthy rewrites. A solid plan is always a good idea before you start.

Using VB and QCard.DLL

You will need to declare all of **QCARD.DLL**'s functions in your Global Module. You can copy them from the VB demo program which is included here. Without these declarations, your program cannot access the routines in the DLL. When you are building your program, you will need a copy of **QCARD.DLL** in your VB directory so VB can access it. Once you have built your program, and you are running it from its own EXE, you will need a copy of **QCARD.DLL** in that EXE's directory. As an alternative, you can put a copy in your Windows directory, in which case Windows can locate it at design time and run time. If your program can't find **QCARD.DLL**, make sure you have it located in the right directory. Once you have finished your project, and you are ready to release it to the world, you will need to ship with it a copy of **QCARD.DLL**.

In many of the sub and function calls to **QCARD.DLL**, you will need to reference the handle of the window in which you are working. In VB, this is the hWnd property of the form you are drawing on, (for example, Form1.hWnd). When drawing and locating items, **QCARD.DLL** uses the MM_TEXT mapping mode. This is a pixel-based coordinate system where the top-left corner of your window is at x, y point 0, 0. Values increase as you move to the right and down. You should set the ScaleMode property of the form you will be working with to Pixel (3), and leave its ScaleTop and ScaleLeft properties at their defaults of 0. This will ensure that both **QCARD.DLL** and your application will be talking in the same terms when making reference to locations.

QCARD.DLL gives you two decks of cards to work with. If you need any more decks than that in one game, then you will have to improvise your own work-around. The cards are referenced by their number. Numbers range from 1 to 104. 1 through 52 is a complete deck and 53 to 104 is the other deck. If you are using just one deck in your game, you can ignore numbers 53 to 104. They will remain unaffected. The easiest way to keep track of what you are doing is to create a Deck array of your own, and fill it with the numbers 1 through 52 (or 104 if you are using 2

decks).

Global Deck (1 To 52) As Integer

Dim i As Integer

For i = 1 to 52

Deck(i) = i

Next i

Then you can go ahead and shuffle the array, if thats what you want.

Randomize Timer

For i = 1 To 10

For j = 1 To 52

k = (Int(Rnd * 52) + 1)

Temp = Deck(j)

Deck(j) = Deck(k)

Deck(k) = Temp

Next j

Next i

You can now make calls to **QCARD.DLL** by referencing your array.

For example,

DealCard Form1.hWnd, Deck(1), xLoc, yLoc

This will deal the first card in your array on Form1 at location xLoc, yLoc in pixel coordinates relative to the top-left corner of your form.

You might want to declare a few Global Constants which will make your job easier. These include the width of the cards and the height of the cards. These values do not change. Another might be an offset value for drawing cards in a row, solitaire fashion, with their left sides aligned. If you want to utilize any of **QCARD.DLL** Block dragging routines (dragging more than one card at a time) you must use an offset of 16 pixels between card tops. Any other value will not work. See the section on dragging routines for more on this.

Global Const CARDWIDTH = 71

Global Const CARDHEIGHT = 96

Global Const OFFSET = 16

Global Const NUMCARDS = 52

Global Const FALSE = 0

Global Const TRUE = NOT FALSE

These will make it easier for you to place your cards where you want them without remembering particular numbers.

QCard.DLL Card Properties

You can think of each card as having a set of properties, some of which you can change and reference as your program executes.

The properties you have access to are as follows:

Related Topics:

- 1) Number
- 2) Color
- 3) Suit
- 4) Value
- 5) X
- 6) Y
- 7) IsVisible
- 8) IsBlocked
- 9) Disabled
- 10) BOOL User1
- 11) int User2, int User3 & int User4

1) Number

You reference a card by its number. The cards are numbered 1 through 104. Two decks worth. To reference a card in a fuction call to **QCARD.DLL**, just call on its number. In the text that follows , nCard refers to the number of the card you are referencing, ie) GetCardSuit(nCard)

2) Color

You can determine a card's color by calling the function

GetCardColor(nCard). Black cards return a value of 1, Red cards return a value of 2.

3) *Suit*

You can determine a card's suit by calling the function

GetCardSuit(nCard) passing it the number of the card you are interested in. Clubs return 1, Diamonds return 2, Hearts return 3, Spades return 4.

4) Value

You can determine a card's value by calling the function `GetCardValue(nCard)`. An ace returns 1, and a King returns 13. The values in between are as you might expect.

5) X

You can determine a card's x location by calling `GetCardX(nCard)`. This will return it's current location from 0 on the left side of your Form. This value is originally set at -72. When a card is dealt using the `DealCard` sub, this value is updated automatically. This value can be set manually by calling `SetCardX(nCard, nLoc)`, giving the sub routine the card number and new x location value.

6) Y

You can determine a card's y location by calling `GetCardY(nCard)`. The value is originally set at -97. When a card is dealt using the `DealCard` sub, this value is updated automatically. It can be set manually by calling `SetCardY(nCard, nLoc)`.

7) *IsVisible*

This is a Boolean value (TRUE or FALSE) which is initially set to FALSE. When a card is displayed using the DealCard sub, it is automatically set to TRUE. It's state can be determined using the function IsCardVisible(nCard) which returns either TRUE or FALSE. It can be set manually with the Sub SetCardVisible(nCard, bValue). For example, SetCardVisible(5, TRUE) sets card 5's visible property to TRUE.

This property comes in quite handy, especially when repainting a window. You can simply call a loop which checks for this property and paints in only those cards which are currently visible:

```
For i = 1 to 52  
  nReturnValue = IsCardVisible(Deck(i))  
  If nReturnValue = TRUE Then  
    DrawCard Form1.hWnd, Deck(i), GetCardX(Deck(i)),  
      GetCardY(Deck(i))  
  End If  
Next i
```

8) *IsBlocked*

The `IsBlocked` property is used when determining if a card is free for dragging or not. When a card is dealt, and another is moved on top of it by you or by the player, this value should be set to `TRUE` for the lower card by calling `AdjustCardBlocked(nCard, TRUE)`. To put it simply, if you have a pile of cards, only the topmost card should have an `IsBlocked` value of `FALSE`. All the cards below it should have an `IsBlocked` value of `TRUE`.

This makes it possible for **QCARD.DLL** to carry out it's hit-testing when initializing a drag event. The `IsBlocked` value of a card is returned by a call to `GetCardBlocked(nCard)`.

9) Disabled

If you want to remove a card from play, you can set it's Disabled property to TRUE. Doing this will make it impossible for the user to select the card for dragging with the mouse. This can be done by calling `SetCardDisabled(nCard, TRUE)`. The card can be enabled again by calling `SetCardDisabled(nCard, FALSE)`.

10) *BOOL* *User1*

This is a Boolean user defined property which you can use as you see fit. You may want to use it to monitor some particular attribute in your game. You can set this value by calling `SetUser1(nCard, bValue)` where `bValue` is either `TRUE` or `FALSE`. You can obtain it's present value by calling the `GetUser1(nCard)` function.

11) *int User2, int User3 & int User4*

User2, User3 and User4 are all integer types. You can set these according to your needs. You set them by calling SetUser2(nCard, nValue), SetUser3(nCard, nValue) and SetUser4(nCard, nValue) where nValue is any integer. You can obtain their current values by calling the GetUser2(nCard), GetUser3(nCard) and GetUser4(nCard) functions. For example, your game might award a player 8 points for placing an Ace in a certain pile. Before shuffling, you could call:

```
For i = 1 To 40 Step 13
```

```
SetUser3(Deck(i), 8)
```

```
Next i
```

Then when the card is played and you are determining how many points to award, you can call nNumPoints = GetUser3(nCurrentCard) and retrieve it.

NOT ALL SMOKE AND MIRRORS

QCard.DLL Subs and Functions

Once you have included the sub and function declarations in your Global Module, you can call **QCARD.DLL** functions just as if you were calling any other function or sub in VB. The following lists 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 all 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.

Initialization Functions

BOOL InitializeDeck (hWnd)
SetCurrentBack(nIndex)
SetDefaultValues()

BOOL InitializeDeck (hWnd)

A call to this function is required for **QCARD.DLL** to operate.

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 "Sorry. Another application is currently using  
        QCard.DLL"  
End  
End If
```

Generally, the InitializeDeck function will only return a FALSE value if the DLL is already in use by another application. Only one application can use **QCARD.DLL** at any given time. This should not pose a problem unless a user tries to run two copies of your game at the same time. It may be a good idea to mention this fact in your game's documentation. When your application ends, **QCARD.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.

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 manipulation of face down cards just like the manipulation you can do with face up cards. The picture is the same for all five cards and is initially set at 1. Call SetCurrentBack(nIndex) where (nIndex) is a number between 1 and 6. You may call SetCurrentBack(nIndex) any number of times to change card backs, 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 goes down 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.

Card drawing subs

DrawCard (hWnd, nCard, nxLoc, nyLoc)

DealCard (hWnd, nCard, nx, ny)

DrawSymbol (hWnd, nValue, nx, ny)

DrawBack (hWnd, nValue, nx, ny)

RemoveCard (hWnd, nCard)

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.

The DrawCard sub does not update any of the card's data members, such as its x or y location, or its IsVisible property. 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 IsVisible property to TRUE.

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 **QCARD.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 present 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.

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.

Information Subs and Functions

GetCardColor(nCard)

GetCardSuit(nCard)

GetCardValue(nCard)

IsCardVisible(nCard) AND SetCardVisible(nCard, bValue)

GetCardBlocked(nCard) AND AdjustCardBlocked(nCard, bValue)

IsCardDisabled(nCard), SetCardDisabled(nCard, bValue)

GetCardX(nCard), GetCardY(nCard), SetCardX(nCard, nValue), SetCardY(nCard, nValue)

GetUser(nCard) & SetUser(nCard, bValue).

GetCardColor(nCard)

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

GetCardSuit(nCard)

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

GetCardValue(nCard)

This function returns the value of the requested card. Aces have a value of 1, Twos have a value of 2, right up to King which has a value of 13. For example:

```
nReturnValue = GetCardValue(nCard)
```

```
If nReturnValue = 11 Then
```

```
Text$ = "Jack"
```

```
End If
```

IsCardVisible(nCard) AND SetCardVisible(nCard, bValue)

This sub and function pair can be used to get and set the IsVisible value for the card. It is initially set to FALSE, and is set to TRUE automatically when the card is dealt using the DealCard sub. You may or may not find this value of use in your application

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. **QCARD.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, **QCARD.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 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 group 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), GetCardY(nCard), SetCardX(nCard, nValue), SetCardY(nCard, nValue)

Use these subs and function to get and set their x and y location properties. These are pixel coordinates based on 0, 0 in the top left corner of the screen. These are very useful for drawing and relocating cards around your window. For example, when the user drags a card 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

SetCardX nSourceCard, nNewx

' you must use 16 pixels as your offset for rows of cards

SetCardY nSourceCard, nNewY + 16

DealCard Form1.hWnd, nSourceCard, GetCardX(nSourceCard),

GetCardY(nSourceCard)

GetUsern(nCard) & SetUsern(nCard, bValue),

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 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 it's position in the array. This makes it easy to move cards from one pile (array) to another as the game executes.

Dragging Functions

InitDrag(hWnd, nx, ny)

AbortDrag()

DoDrag(hWnd, nx, ny)

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

EndDrag(hWnd, nx, ny)

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

ReturnDrag(hWnd, nCard, nxLoc, nyLoc)

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

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 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 either an AbortDrag, an EndDrag or an EndBlockDrag call. nx and ny are the current mouse coordinates.

AbortDrag()

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)

Carries out the drag operation which was initiated by InitDrag call. DoDrag moves the current Source Card to it's new location. 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, 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)

EndBlockDrag ends a block drag operation which was initiated by an InitDrag. 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 it's current location. Return drag can be used for returning cards to their original location after an invalid drag operation.

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.

DRAGGING

Although dragging is made easier using **QCARD.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.

Related Topics:

[A Simple Single Drag Example](#)
[Avoiding 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 run, 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 it's 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 EndDrag 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
```

QCARD.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 it's 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.

Dragging a Block of Cards

QCARD.DLL allows your application to drag a block of cards from one location to another. To carry out this operation, several guidelines must be followed. First, the cards being block dragged must all have the same x location. That is, their left sides must be aligned in column fashion. Second, their y locations must increase in multiples of 16 pixels from the first card in the group being dragged to the last card in the group. When you deal a row of cards, ensure that you offset each card down 16 pixels. Third, the last card in block drag operation is assumed to be the bottom card in a row. If you try to pull the middle three cards from a row and drag them, you will not get the result you want. **QCARD.DLL** drags blocks of cards the same way Windows Solitaire does. When the user presses the mouse button down on the top of a card, that card becomes the first card in the block drag. The last card in the block drag is always the last card in that row.

Related Topics:

[Block Dragging Example](#)
[Avoiding Problems](#)

Block Dragging Example

Like Single Dragging, Block Dragging requires you to provide code for three event procedures for your form: MouseDown, MouseMove and MouseUp. In the MouseDown event procedure, you initialize the drag operation. In the MouseMove event procedure, you carry out the drag operation. In the MouseUp event procedure, you end the drag operation.

Related Topics:

[The MouseDown event procedure](#)

[The MouseMove event procedure](#)

[The MouseUp event procedure](#)

The MouseDown event procedure

Begin the drag procedure by calling the function InitDrag. This function searches through the deck to determine if any card whose IsBlocked property is FALSE is currently located at the mouse's x, y location. If it finds one, it returns the number of that card. This would successfully initialize a single drag operation. If it cannot find an unblocked card at that location, it will determine if the mouse is in the top 16 pixels of any other card, blocked or not. If it finds one, it returns the number of that card. This would initialize a block drag. In the MouseDown event procedure then, a little testing of the selected card's IsBlocked status will tell us if we are about to do a block drag or a single drag:

```
Dim nStatus As Integer  
nSourceCard = InitDrag(Form1.hWnd)  
If nSourceCard = 0 Then  
  ' no card selected  
  AbortDrag  
Else  
  nStatus = GetCardBlocked(nSourceCard)  
  If nStatus = FALSE Then  
    bSingleDragging = TRUE  
  Else  
    bBlockDragging = TRUE  
  End If  
End If
```

The MouseMove event procedure

To drag the block of cards in the MouseMove event you use the sub BlockDrag. You must give this sub a list of the cards you want to drag and the total number of cards in the list. The list itself should be in the form of an array, and you can pass the array to the DLL by referencing it's first element: BlockDrag Form1.hWnd, CardList(0), nNumCards, x, y. By giving the DLL the first element in the array, the DLL will have access to all the elements in the array. If you declare a shared array for this purpose, you can use the Redim statement to resize the array to the correct size when you need it. Then just fill the array with the numbers of the cards you want to drag, beginning with the top card in the block (the Source Card) and ending with the bottom card of the block. See the demo program for a better idea of how this is done.

The MouseUp event procedure

The block drag operation is completed in the MouseUp event procedure by calling the EndBlockDrag function. Again, you must pass this function an array containing the numbers of the cards being dragged and the total number of cards in the list. Since this information does not change between the MouseMove event and the MouseUp event, you can reuse the same array for both, provided the array was declared as Shared:

nDestCard = EndBlockDrag(Form1.hWnd, CardList(0), nNumCards, x,y)

The function returns the number of any card whose IsBlocked property is FALSE which lies beneath the first (Source) card being dragged in the block. You now know the SourceCard and the Destination card and you can go ahead and make comparisons to determine if the drag is valid or not. If it is, you will have to relocate all the cards in your CardList array to their new pile. See the demo program for a better idea of how this might be done.

Avoiding Problems

Problems in block dragging can usually be traced to the array you are trying to pass the BlockDrag sub. If the array is not properly filled beginning with its first element (0), you will end up with strange results. If you pass the sub the wrong number of cards to drag you will also get strange results. Ensure that you are using Redim and that you are filling the array properly if you run into problems.

Another problem you might run into involves the EndBlockDrag function. Let's say for example, you are dragging a block of three cards, 7, 12, and 18. When the mouse button comes up, and the block drag is ended, the EndBlockDrag function goes in search of any unblocked card which overlaps the Source Card (in this case, the Source Card would be card 7). One card that might fit the bill is card 18, since it overlaps card 7 and it is not blocked. To avoid getting this erroneous return value from EndBlockDrag, you can temporarily set the IsBlocked property for the last card in your block drag array to TRUE during your MouseMove event, and remove the block in your MouseUp event after you have ended the block drag operation. See the demo program for an example.

ABOUT QCARDS.DLL

QCARDS.DLL was authored by STEPHEN MURPHY. If you have questions or comments regarding QCARDS.DLL, or if you have any suggestions for improvements for future releases, please contact the author via CompuServe at 70661,2461

Help File Created by:

ANOTHER FINE MESS PRODUCTIONS

(AFMP is a division of Northwest Computer Solutions, Inc.) WE CAN BE CONTACTED THROUGH
DAVID M. SHANK ON COMPUSEVE AT: 70714,546



