CARDPACK Control (ActiveX **version 1.7**)

Properties **Methods** **Events** *Help file revision 1.7.2*

Contents

Thank you for using the Cardpack Control! This small software component gives you a stack of playing cards, giving programmers a powerful object to build card games with. It can be used from Visual Basic, Visual C++ and even on Web pages!

Check out what is new in **this release**.

The most relevant features of this control are:

♥ It comes as an ActiveX control in 16 and 32-bit format. An earlier version (1.1) is still available as a VBX for use in Visual Basic 3.

♠ It can use the card picture library taken from old SOL.EXE or **CARDS.DLL**, which are provided in different versions of MS-Windows, The control is thus very small. If you prefer, you can provide your own card picture library.

♦ It handles (as the name implies) a stack with a virtually unlimited number of cards. Cards can be stacked or spread out in a number of ways.

♣ It has a wide set of game-oriented properties, designed to make it very easy to implement card games with no hassle. These include pattern-based queries and selections in the stack, handy extraction and addition methods, shuffling, sorting, and more!

♥Comes with extensive documentation, programming tutorial, reference information and samples for Visual Basic and Internet Explorer!

This software is *shareware*. Read the **Registration** page for details.

♥♦♣♠

Cardpack Control Installation

Welcome to the Cardpack ActiveX Control installation!

Important

In order to keep the file size down, the distribution archive does not contain the shared Microsoft libraries (DLL) that are needed to run this control:

MFC42.DLL, MSVCRT.DLL	(32-bit version)
OC25.DLL	(16-bit version)

You probably already have these DLLs on your system, but just in case, they are available for download, together with the latest release of the control, at **http://www.multimedia.it/andy/logbook/download** or from in most on-line software archives.

Notes on Licensing

The controls contained in the distribution archives are fully functional. However, you will require a license file in order to get rid of the warning dialog box both at programming time and in executables. See the **Registration** page for details.

Getting Ready for Installation

1. Check out what is new in **this release**.
2. Make sure you understand how to use the different **Variants** of the control.
3. Unpack the distribution archive preserving the directory structure. Use the –d option of *pkunzip* or the *use folder names* when extracting with *WinZip*.

Depending on the package you have downloaded, you may be able to perform an automatic installation or a manual one, as described in the following.

Automatic Installation Procedure

1. If you are running Windows 95, NT 4 or anything better, you can install the software by double clicking on the **SETUP.EXE** program from the Windows Explorer. This will install the controls, help file and sample applications on your system.
2. Check out the Manual Installation Procedure detailed below anyway. Point 3 explains how to install the license file that you may receive separately. Point 5 explains how you can use Visual Basic to check that the controls are properly registered
3. Check the important notes on **CARDS.DLL**.

Manual Installation Procedure

1. Copy the selected control file(s) and in the *System* directory:

   In Windows 3.x, Windows 95 and Windows 98, typically C:\WINDOWS\SYSTEM

   In Windows NT, 16-bit controls in C:\WINNT\SYSTEM, while 32-bit controls in C:\WINNT\SYSTEM32.
2. Copy CARDPACK.HLP in C:\WINDOWS\HELP, C:\WINNT\HELP or in the same directory where you put the control files.
3. The license file should be placed in the \Windows directory. Of course, you should *not* distribute the license file with executable programs you write.
4. If you are a Visual Basic programmer, the CARDPACK.BAS file can be placed in your program source code directory and added to your VB project.
5. Register the control from Visual Basic. In Visual Basic 4, select the *Tools* menu, *Custom Controls*. Click the BROWSE button. Go to the directory where the control is installed, then check it. Close the Controls window clicking OK. In Visual Basic 5, use the *Project*, *Components* menu item.
6. For automatic conversion of Visual Basic programs from VBX to the OCX version of the control, edit VB.INI (normally in C:\WINDOWS); locate the section named

[VBX Conversions16]

and add the following (on a single line):

```
cardpack.vbx={6B17E7E3-AECA-11D0-B4D8-444553540000}#1.0#0;C:\Windows\System\cardpk16.ocx
```

You can do the same for the 32-bit version of the control. The VB.INI section to edit is [VBX Conversions32] and the line to add is:

```
cardpack.vbx={6B17E7E3-AECA-11D0-B4D8-444553540000}#1.0#0;C:\Windows\System\cardpk32.ocx
```

Of course if you installed the controls in other directories than the default C:\Windows\System, you should correct the paths in the two statements above accordingly.

7. Check the important notes on **CARDS.DLL**.

Uninstalling the Software

To uninstall the software, use the Windows Control Panel, "software" applet. In the list you find on the *Add/Remove Software* page, you should see an entry named *Cardpack Control*. Select it and click on the *Add/Remove* button.

♥♦♣♠

Version History

The latest version of the package can always be downloaded from the *CardPack Home Page* at **http://www.multimedia.it/andy/cardpack**,

Version 1.7 (this release)

From this release on, only a 32-bit ActiveX version of the control is provided. Only Visual Basic 5 or higher is supported.

Removed fixed limit on number of cards in a control. The value of **NumCards** is now only limited by system memory.

Added support for 256-colour cards on displays with a palette. The control now looks for a palette in the test card it opens from the DLL (resource #53, the first back) and uses it for all cards rendering.

Fixed a bug where setting a control .Cards property to an empty card set could leave the control with 1 card.

Version 1.6

Migrated 32-bit development environment to VC++ 5 (requires new run-time libraries)

The **NumCards** property can now be set to any number and it will not be rounded to a multiple of 4. This is useful at design time to get a better feel for your layout design.

Added a new method, **DrawCard**, which can be used to perform completely custom layouts or animation effects, as shown in the about box of the sample game included in the archive, Briscola.

The **Shuffle** method was extended to take a *seed* parameter. This allows you to generate predictable random shuffling sequences, so that you can reproduce situations to test your game strategies.

Fixed a bug where registering the control with Windows 95 at set-up could crash on some systems.

Fixed a bug where, on fast computers, shuffling cards was not working very well.

Version 1.5

The control now supports 2 different jokers, as values 14 and 15. They map to card pictures in the library with resources id # 70 and 71, if available, or 70, or fall back to resource 56 if none of those are found.

Now both CARDS16 and CARD32.DLLs have 2 jokers cards (red and blue), as resources 70 and 71, to match the new capabilities in the control.

Fixed a bug where the CARDS16.DLL released in the previous version was broken and could not load the joker.

Fixed a bug where jokers were always added face-down rather than respecting the default stack facing.

Fixed a bug where moving jokers across controls caused an “invalid property value” error. The control was checking that the card had a valid suit, a rule which should not apply to jokers.

Added 2 new properties to extract the top and bottom sections of a card deck without having to loop over each card: **TopCards** and **BottomCards**

Added a '**CurrentCard**' property that removes and returns the card currently under the cursor.

Version 1.4

A patch for making the control work with Visual Basic 5 has been incorporated (formerly released as version 1.3 build 002). The control was crashing VB5 due to some subtle environmental differences with respect to VB4 and other ActiveX containers.

The control code was slightly optimized. Some dead code has been removed.

Fixed a bug where controls with AutoSize= False would not properly reload their size.

Fixed a bug where controls with AutoSize= False were not recalculating their layout correctly at design time.

Fixed a bug where the control was not respecting custom card sizes when reloading properties.

The control now prevents recursion of the Change() event. You can now safely add and remove cards within the event without bothering about implementing a semaphore variable in Visual Basic code.

The CardsLibrary property now always shows the name of currently loaded DLL, even when it is one of the system DLL that are automatically sought by the control. The DLL name now always gets serialized when a project is saved.

If loading the specified cards DLL fails, the control now attempts all the default names, as if no specific DLL had been indicated. This is a workaround against possible problems where loading the project or program on a different system, failing because a differently named cards DLL is present.

Version 1.3

The 32-bit ActiveX version of the control could always be used in Web pages. However, previous versions had a bug where checking a server-side control license from Microsoft Internet Explorer failed. This is now fixed.

The control is not signed yet, so Internet Explorer will warn you against it for security reasons, but it works very well on the Web.

Also, a tutorial about using the CardPack control on **Web Pages** was added to the documentation for all you Internet programmers.

A little bug was fixed that could cause the control to fail loading the Joker card.

Finally, we have included automatic installation and provisions to uninstall the software.

Version 1.2

There is a new property, **CardsLibrary**, that allows you to explicitly specify the cards DLL to be used to provide the pictures for a specific control instance. It is thus possible to freely specify custom cards picture libraries.

Version 1.1

Two OLE controls (OCX, now ActiveX) are now provided: a 16-bit version, CARDPK16.OCX and a 32-bit version, CARDPK32.OCX, with equivalent functionality.

A couple of nasty layout bugs apparent when the stack was shown in "slanted" mode have been evicted.

The binary format for card-set strings has changed and is now different between CARDPK16.OCX and CARDPK32.OCX. This change was a necessity to overcome some peculiarities of 32-bit OLE. Unfortunately it has a direct inpact on application code that loads and saves these strings. See **Loading And Saving** for details.

OCX Version 1.0

A 16-bit OLE control (OCX) version of the control is provided: CARDPK16.OCX, with equivalent functionality to the VBX version 1.1. Using Visual Basic 4, it is possible to automatically migrate existing code that used the VBX version of the control to the new standard.

The control supports new **Methods**, a better alternative to the old **Action** property of the VBX version of the control. Some of the methods also support optional parameters.

A new action/method (**Reset**) is provided.

The control allows the use of a custom graphic as mouse pointer, using the new property **MouseIcon**.

All properties for accessing the components of a **Card Descriptor** are range-checked on setting. An error is raised when an illegal value is used.

VBX Version 1.1

This is the latest existing version of the control in the VBX format, for use with Visual Basic 3.

Fixed bug where adding a card to a position other than Top or Bottom of the stack with **AddCard** was broken.

Fixed bug where adding a card to a stack caused 2 redraws, not 1 when **AutoSize** was set to True.

Some optimization of control repaint has been implemented, in case just one card is inserted or removed. In particular, removing a card does not repaint the whole control unless **AutoSize** =True and **PackOnRemove**=True; adding one card does not repaint the whole control unless **AutoSize** =True.

Because of some code optimisation, the exception caused when reading **SelectedCard** the last time has changed from overflow to invalid property index.

VBX Version 1.0

Initial release

♥♦♣♠

Information

Foreword

This control began its life as a spare time self-training project on custom controls and Windows programming. Day after day, feature after feature, the control became a nice piece of software, and it was really a shame to let it gather dust in a drawer.

This software was never actually intended for commercial use. However, just in case, there is a license mechanism built in that should convince you to check with the author before you attempt to make money out of it. Please read the following sections for details.

See also the **version history**.

I sincerely hope that you have as much fun using this software as I had writing it.

Copyright Notices

The CardPack Control is Copyright © 1995-1997 by Andy Zanna. - All rights reserved.

Windows, Windows NT, Windows 95, Windows 98, Visual Basic, Excel, Access, Office, Visual Basic for Applications, VBScript, Windows Entertainment Packs, Visual C++, Front Page, ActiveX and Internet Explorer are either trademarks or registered trademarks of Microsoft Corporation.

Disclaimer

THIS SOFTWARE AND THE ACCOMPANYING FILES ARE PROVIDED "AS IS" AND WITHOUT WARRANTIES AS TO PERFORMANCE OF MERCHANTABILITY OR ANY OTHER WARRANTIES WHETHER EXPRESSED OR IMPLIED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DAMAGES, INCLUDING ANY LOSS OF PROFITS, LOSS OF DATA, INCLUDING BUT NOT LIMITED TO SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR INDIRECT DAMAGES ARISING FROM THE USE OF THIS SOFTWARE.

Distribution

This software can be freely distributed, provided that:

1) The whole package, as available on the *CardPack Home Page* at **http://www.multimedia.it/andy/cardpack**, is distributed. Specifically, all the files in the package must be included and no changes of any kind must be made to the original files.
2) The distributor acknowledges that this software was designed, implemented and is still owned by the original author, who reserves all rights to it.
3) It is not bundled with a commercial product, with the possible exceptions of "software collections" as clarified below.
4) All the files in the package are included and no changes of any kind are made to the original files.
5) No charge or donation is requested for any copy of this software itself, however made, except for the cost of physical support, transmission medium and time required to perform the copy.

It is explicitly allowed to include the CardPack software distribution package in shareware and freeware collections on networks, BBS and CD-ROMs or other distribution media, where a limited profit margin is taken for the sole sake of supporting the said software archives.

In all other circumstances, written permission must be obtained from the author prior to distribution.

The custom control files (CARDPACK.VBX, CARDPK16.OCX, CARDPK32.OCX) can be distributed alone only as part of a commercial product and only if the author of that product has a valid license for the CardPack software. This license cannot be transferred to users or purchasers of the said commercial product.

User License

This package can be freely used for any legitimate personal, non-commercial purpose.

When used as component of a commercial program, the producer is required to contact the CardPack author in order to obtain permission, in the form of a specific development license file (see

Registration).

The author reserves all right to change the distribution and licensing policy for any future major version of the CardPack software and derived products.

Source code for this package is not freely available. It can, however, be purchased from the author.

About the Author

The author (Andy Zanna) is a computer engineer with a broad experience in software design and production on a wide range of systems, from home computers to real-time systems, enterprise servers and large networks.

This software is in no way related to the author's real work or to the company where he is employed.

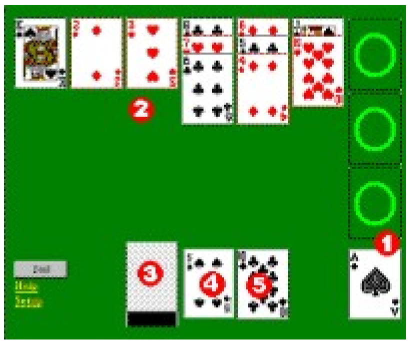
♥♦♣♠

Programming Basics

Overview

This Cardpack control can hold a whole ‘pile’ of cards. In a game, you will typically have several of these piles: the deck, the players’ hands, discarded or captured card piles, cards being played on the table, etc.

Each of these piles will be represented by one instance of the Cardpack control, as shown in the following illustration for a *solitaire*-type game.



Piles of cards (controls) in the game:

(1) 4 piles, one per suit, tightly stacked, holding the sorted cards

(2) 6 piles, spreading down, implementing the actual game board

(3) 1 tightly stacked pile, the ‘deck’, where cards are taken from

(4) 1 single-card pile, holding the card ‘in hand’

(5) 1 stacked pile of discarded cards.

Each pile will be laid out differently: spread, packed, facing up or down, etc. You can set the layout properties of each control to match the function of that card pile, and the control will take care of the rest.

Playing the game means, in essence, moving cards around: from the deck to the players, from the players to the table, from the table to the players or one of the other piles. With the Cardpack Control, you can easily select, pick and move cards from any position in one pile (i.e. one control) to any position in another. You can even move groups of cards in one go.

Implementing a game strategy needs finding cards, sorting them, rearranging them. This control has convenient properties and method for doing all of that.

To learn how to do all of this, read the following sections of the tutorial. For simplicity, the rest of this text assumes that the reader is going to use Visual Basic as a programming environment.

Symbols and Constants

Including the CARDPACK.BAS support file will define all the appropriate symbols and constants you need for Visual Basic programming. Additionally, this file contains a small number of useful support functions.

However, the controls also define all the relevant constants on-line in their type library. Programmers can browse the type library and copy code and constants into their source code using Visual Basic Object Browser (available in Visual Basic 4 or later). For simplicity, this help file will still report examples using the old constants from CARDPACK.BAS.

Data Types

Basically, the control manipulates an array of cards. Each card is a small record of information, called a **Card Descriptor** packed into an integer.

This control has a very large number of properties for manipulating the stack in different ways. You should pay attention to the fact that methods and properties take or return several different data types:

card descriptor	an integer containing all attributes of a single card
index	a 0-based position of some card in the stack.

count	the number of cards matching a certain criterion
set of cards	a binary string containing several packed card descriptors

There is a logic that dictates what kind of data is used and where, but it's easy to get a bit confused, especially since all these data types are just integers and strings to VB and many related properties will have similar names, but take and return different parameters.

A related source of confusion you should watch out for is the presence of 2 different constants:

CARD_EMPTY (0)	This indicates 'no card here' where a card *descriptor* is usually returned (such as in **Card**).
CARD_NONE (-1)	This indicates 'no such card' where an *index* is usually returned (such as in **Find**).

So, in general, read carefully the description of parameters and return values given for each of the control properties and methods in this help file.

♥♦♣♠

Card Pictures

This control is designed with efficiency and flexibility in mind. By default, it will use card pictures from DLLs already installed in MS-Windows. At the same time, it allows you to specify a different picture set. You can control this option using the **CardsLibrary** property.

♥ **Using the standard card library**

When CardsLibrary = "", the control will take the card bitmaps from the bitmap library used by the standard Windows solitaire game: CARDS.DLL or SOL.EXE (sought in this order of preference). Either one or the other of these modules should be available in every installation of MS-Windows, unless they've been removed by the user. See the important notes under **CARDS.DLL**.

These modules provide all the images for a standard 52-card deck, plus a set of back designs and a bit of other stuff. One notable problem is the absence of a picture for a Joker card.

◆ **Using a different card library**

You could provide your own version of CARDS.DLL that has different card designs and (optionally) includes a Joker. Even better, you can specify a specific DLL pathname in CardsLibrary, e.g. CardsLibrary = "c:\mygame\mycards.dll".

See Also

Creating a Card Library

Laying Out the Stack

The control can display a whole stack of cards, according to several layout preferences. The relevant properties are described below.

♥ **Setting card size**

The following properties control the size of all cards in the stack:

CardStretching, **CardWidth**, **CardHeight**

By default CardStretching = False, meaning that all cards have the same, pre-defined width and height as described in the bitmaps loaded from the library. Setting this attribute to True 'unlocks' the CardWidth and CardHeight properties, allowing to specify a different size. Card sizes are expressed in pixels.

◆ **Placing cards on the screen**

These other two properties specify how the card stack is packed or spread on the 'table':

SpreadStyle, **SpreadDir**

Several styles are pre-defined, ranging from 'vertically stacked' to 'spread side-by-side' or 'user-defined'.

The SpreadDir property allows, when using one of the standard spreading styles, to indicate which direction the cards should be spread towards. Choices include up, down, left, right and their diagonal combinations. This property is ignored when SpreadStyle is set to 'User-Defined'.

Setting SpreadStyle to 'User-Defined' 'unlocks' the following properties:

DrawCardEvery

DrawStartX, **DrawStartY**

DrawStepHoriz, **DrawStepVert**

The DrawCardEvery property allows, when showing a tightly packed stack, to display a 'fake thickness' (maybe you don't want a 108 cards stack to be 108 pixels high). The DrawStartX, Y properties indicate where, within the control rectangle, the 1st card should be placed. The DrawStepHoriz, Vert indicate the displacement, in pixels, of every card with respect to the previous one. These properties can be negative, indicating an 'upward' and/or 'leftward' spread.

♣ **Interaction with control size**

Another property that is connected to the stack layout is **AutoSize**. The control computes the card stack size according to the setting of SpreadStyle, SpreadDir, etc. When AutoSize is True, this size is copied into the control size, i.e. the control will size itself according to the number of cards and the 'spreading' settings. When AutoSize is False, the apparent size of the card stack will change, but the control size will remain the same. This means that the control might 'clip out' some of the cards from view or have empty regions.

Note that you can set SpreadStyle to 'user-defined' and still set AutoSize = True, but the control will insist upon deciding the values for DrawStartX, DrawStartY so that the whole stack of card is always visible and fits in the control rectangle. You are still free to set the other layout properties (DrawStepHoriz, DrawStepVert, DrawCardEvery) freely.

♠ **Automatic packing**

By default, cards are laid out evenly. Removing cards from any position in the stack will cause the stack to pack itself up. However, you can prevent the stack from packing itself by setting **PackOnRemove** = False. This will leave 'empty slots' in the stack, so it's best used when the stack is spread out.

♥ **Minimizing refresh**

Automatic packing and sizing also have an impact on the frequency and extent of refreshing in the control. When both are enabled, the control will completely redraw itself on most card insertions and removals. When both are off the control should redraw itself completely only when blocks of cards are inserted or removed. We suggest that you try keeping either or both off to minimise flashing. Normally a reasonable combination is with packing off (you can do it manually with an **Action** when necessary) and auto-sizing on.

See Also

Presetting The Stack

♥♦♣♠

Presetting The Stack

In every card game, you need to load up the deck with a given set of cards before starting to deal. This control has several facilities to make this task as easy as possible.

♥ **Standard decks**

The stack can be pre-set to any number of cards by setting the **NumCards** property to the desired value. The number of cards that you can have in a control only limited by system resources.

The stack will try to fill itself intelligently. In particular, setting the number of cards to 40 or 52 will give you a standard 4-suit 40 or 52-card deck. Setting the value to 54 will give you a 52-card set plus two jokers. Choosing a multiple of 54 will give you a pile with multiple card decks. Odd numbers of cards can be specified as well, but are only useful at design time to better check the game board layout.

The filling logic is as follows:

First, the controls tries to fill in normal cards, starting with ace, 2, 3, ... for all four suits. Next, court cards (King, Queen, Jack) are added. Jokers appear only with multiple of 54 card decks.

At the end of this process, the stack will be sorted in suit-first order (hearts, diamonds, clubs, spades). If the filling algorithm does not suit your needs, just set NumCards to 0, then add the required cards one by one.

♦ **Card facing and back designs**

When adding cards to the stack, they are automatically turned up or down according to the **StackFacing**. property. This also applies when the stack is preloaded with the NumCards property. You can later turn individual cards with the **Facing** property. To flip the whole stack, either change the StackFacing property or set the **Action** property to CARDS_ACTION_TURN_UP.

Depending on the number of cards in the stack (less or more than 54), it will contain one or more 'decks', identified by the design or colour on the card back. These designs are alternatively taken from the two properties **StackBack1** and **StackBack2**. They can be set to the index (1,2, 3...15) of the 2 required back bitmap resources in the DLL..

This is mainly used to replicate the effect of common 2-deck card sets, sold with backs of 2 different colours or designs. For example, if you set NumCards=104, cards 1-52 will have the back indicated by StackBack1, while cards 53-104 will have the back indicated by StackBack2. You can also change these properties later to affect the back design of a whole deck of cards.

To change the back design of individual cards,   you can use the **Back** property. This property lets you specify a different back than those indicated by StackBack1 and StackBack2. You can therefore set a card to be neither part of **Deck** 1 nor 2.

♣ **Empty stacks**

Finally, you can also choose how the stack should look when it's empty, by setting the **StackEmpty** property to one of the predefined 'empty stack' pictures available in the standard DLL, or to none of them, making the stack completely void when empty.

Note that in *design mode*, choosing 'none' for this property will still display a 'dimmed' version of an empty stack picture as a placeholder.

Unfortunately, this version of the control was not designed for handling transparencies. An empty stack or empty regions in a spread stack will not show the underlying form. You can use the standard Visual Basic *BackColor* property to make empty regions match the color of the form.

See Also

Laying Out the Stack

Adding And Removing Cards

In a card game you will be performing lots of insertions, removals, and transfers of cards between stacks. This control has many ways to let you do it.

♥ **Top and bottom cards**

There is a handy set of properties that automatically add to or remove cards from the stack. The two most useful will probably be **TopCard** and **BottomCard**.

Reading one of these two properties will extract a card from the top or bottom of the stack and yield its descriptor. Writing a descriptor to the property will add the card in the appropriate position. Example:

```
' extract a card from under stack 2 and add it on top of stack 1
Stack1.TopCard = Stack2.BottomCard
```

♦ **Adding and removing at an arbitrary position**

Two more properties, **AddCard**(position%) and **RemoveCard**(position%) will insert or remove a card at the appropriate position in the stack. These two properties replace the standard Visual Basic methods *AddItem* and *RemoveItem*, which only work with character strings and are not supported by the control.

♣ **Removing selected cards**

Similarly, the **SelectedCard** property (read-only) will extract the topmost selected card. Reading this property in a loop will extract all selected cards from the stack. Additionally, to quickly remove the card under the cursor without even bothering about card selection, you can use the **CurrentCard** property.

♠ **Iterating insertion and removal**

All these properties can raise a run-time error when they can't be satisfied. For example, SelectedCard will fail with an error when there are no more selected cards. Similarly, TopCard will raise an error if the stack is empty, and so forth. This actually simplifies iterating through a stack. For example, to move all selected cards from one stack to another, you could write something like:

```
On Error Resume Next
Do
  Stack2.TopCard = Stack1.SelectedCard
  If Err Then ...
Loop
```

Note that all stack manipulations described above also trigger a **Change** event on the control and that added cards are automatically turned up or down according to the **StackFacing**. property.

See Also

Block Moves

♥♦♣♠

Manipulating Cards

Once a stack has some cards in it, individual cards can be inspected or manipulated without removing them.

Each card has several *attributes* that can be addressed independently:

Back	index of card back design
Deck	card is part of deck 1, 2 or none, depending on the back design.
Facing	tells if card is facing up or down
Selected	tells if card is currently selected (true or false)
Suit,	hearts, diamonds, clubs, spades
Value,	ace, 2, ... etc.

Additionally, each has a generic **Card** entry that packs all these attributes.

All these attributes are implemented with *property arrays*, indexed on the card number (ranging from 0 to **NumCards**-1). Each attribute yields an encoded integer value called a **Card Descriptor**.

For example, to read the value of the bottom card you could write:

```
c% = Stack1.Value(0)
```

To read all attributes of the topmost card without removing it you could write:

```
c% = Stack1.Card(Stack1.NumCards-1)
```

The values of some descriptors (Suit, Back and Facing) are not simple ordinal values like 0,1,2...., but rather encoded bit patterns. They can be interpreted or assigned to using the constants described in CARDPACK.BAS. For example, to turn the <i>th card face-up you could write:

```
Stack1.Facing(i%) = FACING_UP
```

The **Selected** property array holds Boolean values that can be set to True or False to select or deselect individual cards. Do not confuse this property with the read-only **SelectedCard** property or the **Select** query.

Also note that all stack manipulations described above also trigger a **Change** event on the control.

See Also

Card Descriptor

♥♦♣♠

Querying the Stack

The control has several properties, which we will call "*queries*", that can inspect or manipulate batches of cards in a stack according to a card-matching rule.

The simplest query is perhaps **Count**, which can be used like this:

```
n% = Stack1.Count (0)
```

Despite the fact that this looks to Visual Basic like a read from a property array, it really means "count the cards that match the following pattern" where the pattern "0" matches any card in the stack.

The above example is typically equivalent to:

```
n% = Stack1.NumCards
```

The limit of this syntax is that, if you want to apply a query, you always have to get the result into a variable (like n% above) even if you don't care about it. All four queries below (like Count), return *the number of cards that matched the query*.

The card match pattern used in queries is actually a **Card Descriptor** with any, none or all fields left unspecified. Here are some more examples using the other queries (**TurnUp**, **TurnDown**, **Select**, **Deselect**):

```
n% = Stack1.TurnUp (1)               ' turn face-up all aces. Tell how many were there.
n% = Stack1.TurnDown (SPADES+7)      ' turn face-down all 7 of spades.
n% = Stack1.Select (HEARTS)          ' select all hearts.
n% = Stack1.Deselect (DIAMONDS)      ' deselect all diamonds.
```

For those of you who know about databases, we'll say that the attributes specified in a query are ANDed together. There is no support in the query syntax for OR'ing attributes. Therefore if you want to count, say, hearts and diamonds you have to combine 2 different queries:

```
n% = Stack1.Count (HEARTS + DIAMONDS)                      ' wrong
n% = Stack1.Count (HEARTS) + Stack1.Count (DIAMONDS)       ' right
```

Note that the four queries above are very much like the corresponding **Actions**. However, since Actions do not take a pattern and unconditionally apply to all cards, they are faster than the corresponding query with a 'wildcard' pattern (0).

So, for example, if you need to flip the whole stack, you should use:

```
Stack1.Action = CARDS_ACTION_TURN_UP
```

rather than

```
Stack1.TurnUp (0)
```

Please note that TurnUp and TurnDown also trigger a **Change** event on the control, while Count, Select and Deselect do not. The reason is that turning a card can be an actual game move, while several card selections and deselections may take place before anything really changes on the table.

Two other interesting queries are **Find** and **FindMore**, which also take a card descriptor, but return *a position* (a 0-based *index* of the first card that matched the query):

```
pos% = Stack1.Find (pattern%)        ' find first occurrence of card matching pattern
pos% = Stack1.FindMore (pattern%)    ' find next occurrence of card matching pattern
```

Find will find the 1st card (starting from card 0, at the bottom) which matches the pattern, while FindMore will get any subsequent card that matches. Please note that we could not use "FindNext" as a name for this property because it is a reserved keyword in Visual Basic 3.0.

Additionally, there are two queries that can be use to read or remove in one go block of cards matching a query criterion. These are **GetCards** and **RemoveCards**.

See Also

Card Descriptor
Actions
Block Moves

♥♦♣♠

Block Moves

There is a special group of query and data properties that are meant to move around *blocks* of cards in one go, rather than extracting and adding cards one by one. Using these properties makes common stack manipulations both faster and more pleasant to watch.

All these properties set and get data as Visual Basic strings. However, these strings are merely used as opaque containers for the card data, and therefore you should NEVER try to interpret or manipulate these strings.

Note: *The internal format of these strings can change without notice (and indeed has already changed) between revisions, formats and platform-specific releases of the control. In particular, such format is different between 32-bit OCXs and previous versions.*

♥ First of all, there is a property that represents the whole bunch of cards in a stack. It is called **Cards**, and can be used to quickly copy data across stacks. Example:

```
Stack2.Cards = Stack1.Cards
```

♦ Next, you can use two queries (with the usual pattern matching on the card) to retrieve a set of cards: **GetCards** and **RemoveCards**. The difference is that RemoveCards will remove the matching cards from the source stack, while GetCards won't. Example:

```
MyCards$ = Stack1.RemoveCards (pattern%)
```

♠ Then, you can split a stack in two at an arbitrary card and remove the bottom or top part using the **TopCards** and **BottomCards** properties.

♣ Finally, you can add a block of cards you got with one of the above methods to another stack using the **AddCards** property. Example:

```
Stack2.AddCards = Stack1.GetCards (pattern%)
```

All block moves that modify the stack contents also trigger a **Change** event on the control and that added cards are automatically turned up or down according to the **StackFacing**. property.

See Also

Adding And Removing Cards
Querying The Stack
Loading And Saving

Loading And Saving

Another useful application of **Block Moves** properties is loading and saving a stack of cards.

In 16-bit versions of Visual Basic:

To make sure that all the information is properly saved and restored, it is best to use a *random* file with the *Put* and *Get* commands, rather than *Print #* to a sequential file. This technique guarantees that the binary string that describes a stack contents is properly delimited in a file record.

Example:

Suppose you have a game program with 3 card stacks, called Player(0), Player(1) and Deck:

```
' Save players and deck to file
Sub Game_Save ( )
    Dim S$
    Open "game.sav" For Random As #1
    S$ = Player(0).Cards
    Put #1, , S$
    S$ = Player(1).Cards
    Put #1, , S$
    S$ = Deck.Cards
    Put #1, , S$
    Close #1
End Sub

' Load players and deck from file
Sub Game_Load ( )
    Dim S$
    Open "game.sav" For Random As #1
    Get #1, , S$
    Player(0).Cards = S$
    Get #1, , S$
    Player(1).Cards = S$
    Get #1, , S$
    Deck.Cards = S$
    Close #1
End Sub
```

In 32-bit versions of Visual Basic:

The internal format of card strings is different between 32-bit OCXs and previous versions. Do not assume that these formats will remain the same in future versions of the control. Continue to treat card-set strings as ‘‘opaque’’ data.

Yes, due to some constraints of 32-bit OCX development environment, card strings now contain *text* data, rather than binary. This has 2 major consequences:

1. Your loading and saving code should use *text* files, rather than binary, opening the file in *Input* or *Output* mode, rather than *Random*, and using *Print #* and *Input #* statements as shown in the example below.
2. You will not be able to load card strings saved from a 16-bit OCX into a 32-bit OCX.

Example:

```
' Save players and deck to file
Sub Game_Save ( )
    Dim S$
    Open "game.sav" For Output As #1
    S$ = Player(0).Cards
    Print #1, S$
    ‘.... etc.
    Close #1
End Sub

' Load players and deck from file
Sub Game_Load ( )
    Dim S$
    Open "game.sav" For Input As #1
    Input #1, S$
    Player(0).Cards = S$
    '.... etc.
    Close #1
End Sub
```

See Also

Querying The Stack

Block Moves

♥♦♣♠

Methods

The control provides a set of methods that let you affect *all* cards in the stack at once. The first two methods you are likely to use in your games are perhaps **.Sort** and .**Shuffle**.

The methods **.SelectAll**, **.DeselectAll**, **.Turn** should be pretty obvious. They mimic the corresponding queries, with the difference that they apply to all cards. So:

```
Stack1.SelectAll
```

is practically equivalent to (and faster than):

```
n% = Stack1.Select(0)
```

Another interesting method is **.Pack**, that can be used to pack tightly a stack that had some cards removed when **PackOnRemove** was set to False.

In summary, the control defines the following methods:

Clear	Makes the stack empty. It is equivalent to setting Stack.NumCards = 0
Reset	Reset stack as if just created from the Visual Basic Toolbox
Shuffle *seed*	Shuffle the stack. The optional seed parameter allows you to generate predictable random shuffling sequences, so that you can reproduce situations to test your game strategies. If not specified, the Windows millisecond timer is used to seed the random number generator.
Sort *order*	Sort cards according to specific order, or (when the optional parameter is not specified) according to the stack default order. The *order* parameter values are 0 (suit-first) and 1 (value-first) as in the **Sorting** property.
Turn *direction*	Turn all cards in specific or or (when the optional parameter is not specified) according to the stack default facing direction. The *direction* parameter values are 0 (face down) and 1 (face up) as in the **StackFacing** property.
SelectAll	Unconditionally select all cards
DeselectAll	Unconditionally deselect all cards
Pack	Pack cards tight, closing empty slots where cards were removed
DrawCard	Draw a single card on form at arbitrary position. Use for animations

Example:

```
Stack.Clear
Stack.Turn
```

See Also

Actions

Querying The Stack

♥♦♣♠

Actions

Actions are commands that affect ALL cards in the stack at once. They were used in the VBX version of the control to work around an architectural limitation: VBXs cannot define custom **Methods**. Instead, they have an **Action** property which can be set to an 'action code', triggering the command.

So, with the VBX version of the control you would write:

```
Stack1.Action = CARDS_ACTION_SORT
```

rather than the following, which only works with the OCX control:

```
Stack1.Sort
```

For compatibility reasons, actions have been retained even in the ActiveX version of the control. However, they should be considered obsolete: you are strongly encouraged to use Methods instead.

See Also

Methods

♥♦♣♠

Card Selection

The control has several features for managing the selection of cards. There are two main strategies to handle selection:

- ♥ **Automatic Selection**

  By default, clicking on a card in the stack does not select it. This is needed because sometimes you'll just want to drag or extract the card right away without highlighting it first.

  Setting **AutoSelect**=True will reverse this behaviour and make the stack behave more or less like a ListBox control. Clicking a card will automatically select it (the card will be highlighted). Clicking it again will de-select the card. Setting **MultiSelect**=True will turn on multiple selection (as happens with ListBoxes).

- ♦ **Manual Selection**

  Additionally, it is possible to select or deselect one, several or all cards from program code, using either a) the individual card **Selected**(index%) attribute or b) the **Select**(pattern%) and **Deselect**(pattern%) queries or c) the SELECT and DESELECT **Actions**.

Once one or more cards are selected, it is possible to extract all the selected cards, one by one, using the **SelectedCard** property.

The **LastSelected** property will return the index of the last selected card. Note that this property can return -1 (CARD_NONE) if there is no selected card in the stack or (in a multi-select situation) the card has just been de-selected. As such, this property is of limited use when the stack is set to multi-select. In any case there is another property, called **Current**, which returns the index of the card that is right under the mouse when the most recent event was detected, regardless of its selection status. For example, the following code:

```
Sub Stack1_MouseDown (Button As Integer, Shift As Integer, X As Single, Y As Single)
  Dim i%
  i% = Stack1.Current
  Stack1.Selected (i%) = Not Stack1.Selected (i%)
Sub End
```

simulates the behaviour of the control when AutoSelect=True.

Tip

To retrieve or remove all selected cards from a stack in one go, you can use one of the following query commands:

```
c$ = Stack1.GetCards(CARD_SELECT_BIT)
c$ = Stack1.RemoveCards(CARD_SELECT_BIT)
```

See Also

Dragging Cards

♥♦♣♠

Dragging Cards

How do you show cards moving on the form from stack to stack, either automatically or in a "drag and drop" fashion?

Using the standard Visual Basic methods, properties and events that control drag-and-drop typically will not give you what you are looking for.

First of all, when using the standard Visual Basic drag-and drop, you can only use an *icon* as the dragged object image, not the real picture of one or more cards.

Second, setting AutoDrag=True causes all mouse actions to be trapped by the Visual Basic run-time, preventing the control from selecting the card under the mouse or even detecting which card is under the mouse at the time the drag is started.

Even if you can live with Visual Basic's drag and drop, it is more useful to leave AutoDrag set to False, handle mouse clicks normally and start a drag with the .Drag method.

If you want to show a real card picture moving on the surface of your form, there are two basic ways of doing this with the Cardpack control

1) Use a spare Cardpack control to hold the cards you are dragging and move that control on the window in sync with the mouse.
2) Use the **DrawCard** graphical method to draw directly on the form, also synchronized with mouse movements.

Here are the detailed instructions for the first method:

Prepare a spare, hidden, empty stack of cards. When you are about to start a drag, move the selected cards to the hidden stack, then move the hidden stack at the position where you want the drag to start and make it visible.

Suppose stack Cards2 has MultiSelect = False and AutoSelect = True. Suppose a spare, hidden stack exists, called 'DragCards'. This is what you would do:

```
Sub Cards_MouseDown (Button As Integer, Shift As Integer, x As Single, y As Single)

    On Error Resume Next                                    ' prevent exception if no cards get selected
    DragCards.TopCard = Cards.SelectedCard          ' move selected card to drag stack

    If DragCards.NumCards > 0 Then                        ' make DragCards appear on top of Cards
        DragCards.Left = Cards.Left
        DragCards.Top = Cards.Top
        DragCards.ZOrder 0                                       ' make sure it shows on top of all controls
        DragCards.Visible = True                               ' and then show it.
    End If
End Sub
```

On subsequent mouse move events, Move the spare stack so that it stays centered under the cursor. Note that the co-ordinates we're getting are relative to 'Cards' (the control that has the capture), not 'DragCards'.

```
Sub Cards_MouseMove (Button As Integer, Shift As Integer, x As Single, y As Single)
    If DragCards.Visible = True Then
        DragCards.Move x + Cards.Left - DragCards.Width \ 2, y + Cards.Top - DragCards.Height \ 2
    End If
End Sub
```

When the mouse button is released, hide again the spare stack and move its cards to the target.

```
Sub Cards_MouseUp (Button As Integer, Shift As Integer, x As Single, y As Single)
    If DragCards.Visible = True Then
        DragCards.Visible = False
........' check what area of the board you are dropping onto, find out who is CardsDest
```

```
            CardsDest.TopCard = DragCards.TopCard
    End If
    End Sub
```

Be aware that this method has a couple of drawbacks:

First of all, performance can be bad on some PC's, which will display cards 'staggering' across the screen. You can somewhat minimise this effect by making the cards in the dragged stack smaller than the standard size.

Second, you are not told what control you are dropping the cards onto: You have to check the mouse co-ordinates and decide for yourself.

If you can live with some more complexity and are willing to call the Windows API, you can use the same event structure, but draw directly the card pictures on the form with the **DrawCard** method. You don't need to use a spare Cardpack control in this case. The real tricky part is that you have to take care of saving and restoring the background while moving the card across the form, a job that is best done calling the Windows API directly. In return, you can get some really smooth card animations.

Note that, if you want to draw *over* all the controls on the form, you have to set the form's **ClipControls** property to False.

♥♦♣♠

DDE

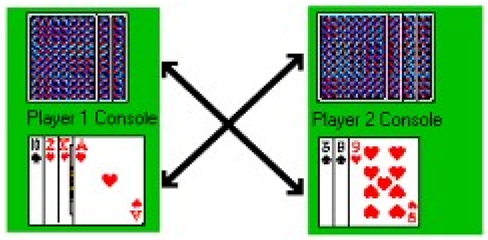
This object fully supports DDE links as client and server. All the standard Visual Basic properties and events are provided.

In particular, it is possible to link 2 stacks so that the second is the spitting image of the first one. This feature makes it incredibly simple to write multi-user version of card games that work over a Network-DDE capable LAN, like those provided with Windows For Workgroups 3.11, Windows 95 or NT.

Note: *on Windows 95, only 16-bit Network-DDE is supported. Only Windows NT appears to support 32-bit Network DDE.*

The control will send or get the full stack contents in either binary or text (tab-separated hex card descriptors) format.

The rule that added cards are automatically turned up or down according to the **StackFacing** property also applies to DDE transfers. This means that you can link a down-turned stack to an up-turned stack and vice-versa, to create a two-player game, as shown in the following image:



Also note that it is safe to cross-link two controls (i.e. have them to link to each other). By default, This control will not send back via DDE data that it has just received via DDE. This prevents the two linked controls to enter an endless update loop. If, for some reason, you need to change this behaviour (e.g. maybe you have 3 controls linked in a chain), just set the **LinkEcho** property to True.

♥♦♣♠

Card Descriptor

Overview

The control maintains information for each card in compact (16-bit), binary form called a *Card Descriptor*.

This is very efficient both inside the control and for client programs, which can move and manipulate cards very quickly. The down side is that this binary description is not very programmer-friendly, but can be managed using the pre-defined constants supplied in the support file. Moreover, several properties are supplied that can return a specific attribute of a given card, saving you the need to fiddle with individual bit-fields.

Thus, a *Card Descriptor* is a 16-bit integer that packs suit, value, facing and back type for a card. Several properties in the control accept and return CardDescriptors. The most commonly used is perhaps **Card**.

```
Card% = Cardpack1.Card(0)            ' inspect bottom card
```

The coding scheme reserves some bits for each attribute. The value '0' is reserved for all attributes, which start their numbering at 1. For convenience, the card value is encoded in the lowest significant bits, allowing to express card face values as ordinary integers (i.e. ACE=1).

It is thus possible to:

- 'mask out' any of the properties to extract them individually or in groups;
- Combine attributes with a simple integer sum;
- Use a '0' value in an attribute field to indicate 'unspecified' when a CardDescriptor is used in a query.

Details

We need the following range of values to fully identify each card and to keep track of its selection status:

Value	1..15	(Ace,2,3,...Jack, Queen, King, Joker1, Joker2)
Suit	1..4	(Hearts, Diamonds, Clubs, Spades)
Facing	1..2	(Up,Down)
Back	1..13	(as many as are available in the image library)

As already indicated, note that the values for all attributes start at 1, not 0. '0' is a reserved value that means 'don't care' when looking for cards in the deck.

This yields 4+3+2+4 = 13 bits of information, which are stored internally in 2 bytes. If we specify constants for Suit, Facing and Back values, we can map these values in the following ranges (hex values shown):

Value	0001..000F
Suit	0010..0040
Facing	0100..2000
Back	1000..D000

Additionally, the bit at hex 0800 is used as a selection flag. The **Deck** attribute, instead, is computed by comparing the Back attribute with the current setting of the **StackBack1** and **StackBack2** control properties.

A single card can thus be identified as an Integer variable holding a bit mask in the form

```
CardDescriptor := {Value + Suit + Facing + Back}
```

such as:

```
Card% = (MY_FAVOURITE_BACK+FACING_DOWN+SPADES+KING)
```

It is thus possible, when looking for cards in a deck, to leave some attributes unspecified, e.g.:

```
Index% = Cardpack1.Find(SPADES)
```

See Also

Manipulating Cards

♥♦♣♠

Properties

The control supports the non-standard properties listed in the following table:

AddCard	**Current**	**GetCards**	**SpreadDir**
AddCards	**CurrentCard**	**LastSelected**	**SpreadStyle**
AutoSelect	**Deck**	**LinkEcho**	**StackBack1**
AutoSize	**Deselect**	**MultiSelect**	**StackBack2**
Back	**DrawCardEvery**	**NumCards**	**StackEmpty**
BottomCard	**DrawStartX**	**PackOnRemove**	**StackFacing**
BottomCards	**DrawStartY**	**RemoveCard**	**Suit**
CardHeight	**DrawStepHoriz**	**RemoveCards**	**TopCard**
Card	**DrawStepVert**	**Select**	**TopCards**
Cards	**Facing**	**Selected**	**TurnDown**
CardStretching	**Find**	**SelectedCard**	**TurnUp**
CardWidth	**FindMore**	**Sorting**	**Value**
Count			

Methods

The Cardpack control supports the following non-standard methods:

Clear	**Sort**	**DeselectAll**
Reset	**Turn**	**Pack**
Shuffle	**SelectAll**	**DrawCard**

LinkEcho

This property is used to control whether the control will echo DDE data it received from a server to other DDE clients linked to itself.

One of the main reasons you may want to use DDE with this control is creating network games where controls are cross-linked. For this reason, by default this property is set to False. This lets you safely cross-link controls and not worry that they will start ping-ponging data between themselves in an endless loop.

There maybe some uncommon situations where you want to enable echoing: for example, you have a chain of 3 controls linked together so that control #2 is to forward DDE data it received from control #1 to control #3.

See Also

DDE

Card

A property array   that returns the **Card Descriptor** for the <i>th card in the stack, i.e. all card properties packed together, except selection status. While scanning the stack it is also possible to set and get individual attributes for a single card using the other 'Card' property arrays.

Example:

```
' Set full attributes of a card in one go
Player1.Card(0) = ACE + SPADES + FACING_DOWN + CARD_LAST_BACK
```

See Also

Manipulating Cards

♥♦♣♠

Current

This read-only property returns the index of the card that is right under the mouse when the most recent event was detected, regardless of its selection status.

On many occasions this is more convenient than reading the **LastSelected** property. If you don't need to check the card and you can remove it right away, you may want to use the **CurrentCard** property instead.

This property will return CARD_NONE if no card is under the cursor.

Example:

```
' on mouse click, drop card on table
Table.TopCard = Player1.RemoveCards(Player1.Current)
```

See Also

Card Selection

♥♦♣♠

Back

A property array that returns or sets the 'back' field of the **Card Descriptor** for the ith card in the stack. Legal values are in the range (as defined in CARDPACK.BAS):

Constant	Value
CARD_FIRST_BACK	&H1000
CARD_LAST_BACK	&HF000

Example:

```
' Show a fancy back
Cards1.Facing(Cards1.NumCards-1) = FACING_DOWN
Cards1.Back(Cards1.NumCards-1) = CARD_LAST_BACK
```

See Also

Manipulating Cards

♥♦♣♠

Facing

A property array   that returns or sets the 'facing' field of the **Card Descriptor** for the ith card in the stack. The value should be one of (as defined in CARDPACK.BAS):

Constant	Value
FACING_UP	&H100
FACING_DOWN	&H200

Example:

```
' Show a fancy back
Cards1.Facing(Cards1.NumCards-1) = FACING_DOWN
Cards1.Back(Cards1.NumCards-1) = CARD_LAST_BACK
```

See Also

Manipulating Cards

♥♦♣♠

Deck

A property array that returns or sets the 'deck' to which the ith card belongs by comparing the **Back** property with the **StackBack1** and **StackBack2** properties.

When read, this property will return:

1	card Back matches StackBack1, so this card is in deck 1
2	card Back matches StackBack2, so this card is in deck 2
0	card Back matches none of them, so it's neither in deck 1 or 2.

When written to, this property will (according to the value):

1	set Back to match StackBack1, making card part of deck 1
2	set Back to match StackBack2, making card part of deck 2

The logic of this property is respected for any number of cards, but is only really useful in stacks with 2 x 54 or 2 x 52 deck of cards.

Example:

```
If Cards1.Deck(0) <> Cards1.Deck(1) Then debug.print “mixed deck!”
```

See Also

Manipulating Cards

Presetting The Stack

♥♦♣♠

Suit

A property array that returns or sets the 'suit' field of the **Card Descriptor** for the ith card in the stack. The value should be one of (as defined in CARDPACK.BAS):

Constant	Value
HEARTS	&H10
DIAMONDS	&H20
CLUBS	&H30
SPADES	&H40

Example:

```
If Cards1.Suit(0) = SPADES Then debug.print “got some spades”
```

See Also

Manipulating Cards

♥♦♣♠

Value

A property array that returns or sets the 'value' field of the **Card Descriptor** for the ith card in the stack. The value should be one of (as defined in CARDPACK.BAS):

Constant	Value
ACE	1
2..10	2..10
JACK	11
QUEEN	12
KING	13
JOKER1	14
JOKER2	15

Example:

```
' Check if we were lucky picking up the latest card
If Cards1.Value(Cards1.NumCards - 1) = JOKER1 Then debug.print “horray!”
```

See Also

Manipulating Cards

♥♦♣♠

Selected

A BOOL property array, returns TRUE if the ith card is selected (like in ListBoxes). Can be assigned to modify the selection status of   a single cards. If more than one card needs to be selected according to a specific criterion, you may want to use the **Select** query. If all cards must be selected or deselected in one go, you can use the appropriate **Actions.**

Example:

```
' select topmost card
Cards1.Selected(Cards1.NumCards - 1) = True
```

See Also

Manipulating Cards

Card Selection

Actions

♥♦♣♠

AutoSelect

Setting this property to True will make the stack behave more or less like a ListBox control. Clicking a card will automatically select it (the card will be highlighted). Clicking it again will de-select the card.

If a card is selected, the **LastSelected** property will return the *index* of that card. There are several other ways to inspect or extract selected cards, which are explained in the programmer's guide.

This property works together with **MultiSelect**, which, when True, will turn on multiple selection (as happens with ListBoxes).

Example:

```
Cards1.AutoSelect = False
```

See Also

Card Selection

♥♦♣♠

MultiSelect

Turn on multiple selection for this stack. If you also have set the **AutoSelect** property to True, the card stack will behave like a multiple-selection ListBox control: it will be possible to toggle the selection status of individual cards in the stack with the mouse.

Note that the **LastSelected** property is not very useful in this case, because the control tracks the *status* of cards, rather than the selection history. Therefore, if you *deselect* a card, it is impossible for the control to decide which of the other selected cards become the 'last selected'. You can still use one of the several other ways explained in the programmer's guide to inspect or extract selected cards.

Example:

```
Cards1.MultiSelect = True
```

See Also

Card Selection

♥♦♣♠

SelectedCard

This property extracts from the stack the topmost selected card and returns its **Card Descriptor**. It can be used in a loop to extract all selected cards, one by one. If the stack has no selected cards, an exception is raised.

Example:

```
On Error Resume Next
Do
  Stack2.TopCard = Stack1.SelectedCard
  If Err Then ...
Loop
```

See Also

Card Selection

♥♦♣♠

LastSelected

Returns the *index* of the most recently selected card, or -1 (CARD_NONE) if none was selected. Can be checked from click events, and is unset if the deck is reset to a different number of cards or if the selected card is removed.

Please note that, in a stack with **MultiSelect**=True, this property may return -1 (CARD_NONE) when a de-selection occurred, even if there are some selected card in the stack. In a multi-selectable stack you may find that the **Current** property is more useful.

Example:

```
Table.TopCard = Player1.RemoveCard(Player1.LastSelected)
```

See Also

Card Selection

♥♦♣♠

Count

Query the deck for the number of cards matching certain criteria. Returns the total count of cards in the deck that match the criterion.

Like all queries, it uses a syntax similar to accessing a property array, but takes a query pattern in place of an index.

```
HowMany% = Player(1).Count(ACE)
```

Please note that doing

```
HowMany% = Player(1).Count(0)
```

will ALWAYS return the real number of cards in the stack, even if the stack has some 'empty' slots. You can use this trick in place of **NumCards** when you have disabled **PackOnRemove**.

Example:

```
' Counting spades in the simplest way
n% = Cards1.Count(SPADES)
```

See Also

Querying the Stack

♥♦♣♠

Find

Query the stack for a card matching a certain criterion. Returns the index of first matching card.

Like all queries, it uses a syntax similar to accessing a property array, but takes a query pattern in place of an index.

To continue the search from the last card that matched, use **FindMore**.

Example:

```
' Moving all Spades from deck 1 to deck 2 one by one
Do
   i% = Deck1.Find(SPADES)
   if i% = CARD_NONE Then Exit Loop

   Card% = Deck1.RemoveCard (i%)
   Deck2.AddCard(-1) = Card%
Loop
```

See Also

Querying the Stack

♥♦♣♠

FindMore

Queries the stack for the 'next' cards that matches the criterion, searching onward from the last **Find** or **FindMore**.

Like all queries, it uses a syntax similar to accessing a property array, but takes a query pattern in place of an index.

Example:

```
' Counting spades in a very elaborate way
n% = 0
I% = Deck1.Find(SPADES)

If I% <> CARD_NONE Then
   n% = 1
   Do
      I% = Deck1.FindMore(SPADES)
      if I% = CARD_NONE Then
           Exit Loop
      Else
                n% = n% +1
    Loop
End If
```

See Also

Querying the Stack

♥♦♣♠

Select

Queries the stack for cards matching a criterion and marks them as selected. Does not affect other cards. Returns the number of cards that matched.

Like all queries, it uses a syntax similar to accessing a property array, but takes a query pattern in place of an index. To select all cards in one go, you can also use the appropriate **Actions**

Example:

```
n% = Select (ACE)
```

See Also

Querying the Stack

Card Selection

Actions

♥♦♣♠

Deselect

Queries the stack for cards matching a criterion and marks them as deselected. Does not affect other cards . Returns the number of cards that matched.

Like all queries, it uses a syntax similar to accessing a property array, but takes a query pattern in place of an index. To select all cards in one go, you can also use the appropriate **Actions**

.

Example:

```
' these 2 are equivalent
n% = Stack1.Deselect (0)
Stack1.Action = CARDS_ACTION_DESELECT
```

See Also

Querying the Stack

Card Selection

Actions

♥◆♣♠

TurnUp

Queries the stack for cards matching a criterion and turns them face-up. Does not affect other cards. Returns the number of cards that matched.

Like all queries, it uses a syntax similar to accessing a property array, but takes a query pattern in place of an index. To turn all cards in one go, you can also use the appropriate **Actions**

.

Example:

```
' show all aces (if any
n% = Stack1.TurnUp (ACE)
```

See Also

Querying the Stack

Actions

♥♦♣♠

TurnDown

Queries the stack for cards matching a criterion and turns them face-down. Does not affect other cards. Returns the number of cards that matched.

Like all queries, it uses a syntax similar to accessing a property array, but takes a query pattern in place of an index. To turn all cards in one go, you can also use the appropriate **Actions**

.

Example:

```
' these 2 are equivalent
n% = Stack1.TurnDown (0)
Stack1.Action = CARDS_ACTION_TURN_DOWN
```

See Also

Querying the Stack

Actions

♥♦♣♠

PackOnRemove

When this property is True (the default situation), the stack is packed whenever one or more cards are removed. There will never be 'empty slots' in the stack and the **NumCards** property will always return the exact number of cards in the stack.

When PackOnRemove = False, removing one or more cards from the middle of the stack will leave 'empty slots'. This has one notable side effect: the NumCards property will return the 'number of slots' (including empty ones) rather than the real number of cards in the stack.

There is an easy work-around: you can just ask the stack to count its cards with the **Count** query:

```
RealNumCards% = Stack1.Count (0)
```

Moreover, inspecting the contents of a single card (using **Card**(n%), **Value**(n%), etc.) with an index that correspond to an empty slot will return CARD_EMPTY (0).

Please understand that PackOnRemove=False is meant to provide a better visual feedback when cards are removed, not as a general-purpose layout technique. At some point you may want to pack the stack using Cards.**Action** = PACK to avoid getting confused by the side-effects.

However, all properties except NumCards should work fine even when the stack has empty slots, including queries and selections.

Example:

```
' Prevent full redraws when removing cards, but still allow stack to grow
Stack.PackOnRemove = False
Stack.AutoSize = True
```

See Also

Adding And Removing Cards

Laying Out the Stack

♥♦♣♠

AddCard

Adds (actually inserts) the specified card to the stack in the specified position. The argument passed in the fake property array is taken as the position index, while a **Card Descriptor** should be specified as the value in the assignment.

Specifying -1 for the position adds the card to the top of the stack (alternatively, the **TopCard** property can be used for this purpose).

Note that this assignment can generate a run-time error if the maximum number of cards in the stack is exceeded or if the index specified is more than 1 above the current number of cards.

Example:

```
Stack.AddCard(5) = ACE+ SPADES + BACK_1     ' inserts an ace of spades as 5th card
```

See Also

Adding And Removing Cards

♥♦♣♠

RemoveCard

Returns the **Card Descriptor** of the card specified as index. The card is removed from the stack. This property replaces the standard VisualBasic RemoveItem method.

Please note that the **TopCard** and **BottomCard** properties can be used as a shortcut for retrieving and removing the topmost and the bottom card.

Example:

```
Card% = Stack.RemoveCard(0)                    ' remove the bottom card from the stack
```

See Also

Adding And Removing Cards

♥♦♣♠

TopCard

Removes the topmost card and returns its **Card Descriptor**. Can also be used for adding cards on top of the stack, by assigning a card descriptor to the property.

Example:

```
' Reading this property...
Card% = Stack.TopCard

' ...is equivalent to:
Card% = Stack.RemoveCard(Stack.NumCards-1)

' Setting this property...
Stack.TopCard = Card%

' ...is equivalent to:
Stack. Stack.AddCard (-1) = Card%
```

See Also

Adding And Removing Cards

♥♦♣♠

BottomCard

Removes the bottom card from the stack and returns its **Card Descriptor**. Can also be used for sliding cards under the stack, by assigning a card descriptor to the property.

Example:

```
' Reading this property...
Card% = Stack.BottomCard

' ...is equivalent to:
Card% = Stack.RemoveCard(0)

' Setting this property...
Stack.BottomCard = Card%

' ...is equivalent to:
Stack.AddCard(0) = Card%
```

See Also

Adding And Removing Cards

♥♦♣♠

NumCards

Returns or sets the number of cards in this stack. The value can be set both at design and run-time. During a game, this number will automatically vary depending on the number of cards that have been added and removed from the deck.

If this value is explicitly assigned to (either at design or run-time), this action is interpreted as a 'reset' to the stack, which is re-loaded with a sorted deck of cards. The control will try to load an even number of cards for each suit, adding court cards and jokers if possible.

Note that, at run-time, only multiples of 4, or multiples of 54 will give you sensible stacks. Setting this property to odd numbers (3,5,7) is, however, useful for getting a better feeling for the layout at design time.

In theory, the number of cards held by a control is only limited by the system resources. The control is designed allocate memory for the cards in blocks, thus limiting the number of allocations and using memory very efficiently. In your games, the controls should very quickly settle on the best memory size for the number of cards they are likely to hold.

By default, this property is set to 40.

Here are some examples of the effects of loading NumCards with different values:

Value	Effect
0	Empty deck
4	Poker of Aces
40	4 suits: Ace..7, J,Q,K
52	4 suits: Ace..10, J,Q,K
54	Like 52, + 2 jokers
104	2 decks of 52 cards
108	2 decks of 54 cards, each with 2 jokers

If you need to count the number of cards in the stack that match a specific criterion, you can use the **Count** query.

There is another time where you may want to use Count rather than NumCards for reading the number of cards in the stack. This happens when the **PackOnRemove** property is set to false. In that case, the NumCards property returns the number of 'slots' in the deck, including 'empty' cards.

Example:

```
' start new game
Stack.NumCards = 52
Stack.Action = CARDS_ACTION_SHUFFLE
```

See Also

Presetting The Stack

Querying the Stack

StackEmpty

Determines what picture (if any) is displayed for an empty stack.

Possible choices are 0-None (no cards or placeholders will show when the stack is empty) or one from the 2 pre-defined placeholders available in the standard library. These placeholders show a green card-size rectangle with either a cross or a circle.

The following table lists the legal values, as defined in CARDPACK.BAS:

Constant	Value	Meaning
CARDS_EMPTY_NONE	0	Nothing is shown for an empty stack.
CARDS_EMPTY_CROSS	1	A card placeholder with a cross is shown
CARDS_EMPTY_CIRCLE	2	A card placeholder with a circle is shown

By default, this property is set to 0 (CARDS_EMPTY_NONE), making the stack invisible when empty.

Example:

```
' make the cards disappear completely
Cards.StackEmpty = CARDS_EMPTY_NONE
Cards1.NumCards = 0
```

See Also

Laying Out The Stack

♥♦♣♠

StackFacing

Determines the default facing direction of the stack, i.e. the direction cards will be facing when added to the stack. This includes presetting the stack with the **NumCards** property, adding cards with **Block Moves**, copying stacks and receiving card data via **DDE**.

You can later turn individual cards with the **Facing** property. To flip the whole stack, either change the StackFacing property or set the **Action** property to CARDS_ACTION_TURN_UP.

The following table lists the legal values, as defined in CARDPACK.BAS:

Constant	Value	Meaning
CARDS_FACING_DOWN	0	Added cards will face down by default.
CARDS_FACING_UP	1	Added cards will face up by default

By default, this property is set to 0 (CARDS_FACING_DOWN).

Example:

```
' show 4 aces, face up
Cards.StackFacing = CARDS_FACING_UP
Cards1.NumCards = 4
```

See Also

Presetting The Stack

♥♦♣♠

Cards

This property can be used to quickly retrieve or set the whole contents of the stack. It returns a String containing the card binary data. It does not affect the source stack. This property can effectively be used to *copy* stacks.

This is the *default property* for the control.

Example:

```
' copy one stack to another taking advantage of the default property
Stack2 = Stack1
```

See Also

Block Moves

♥♦♣♠

AddCards

This property can be used to quickly add a bunch of cards to the stack. It takes a String containing the card binary data.

Example:

```
DroppedCards.AddCards = Player1.Cards
```

See Also

Block Moves

Querying the Stack

♥♦♣♠

GetCards

Query the deck for all the cards matching certain criteria. Returns a String containing the card binary data. It does NOT remove the matching cards from the stack.

Like all queries, it uses a syntax similar to accessing a property array, but takes a query pattern in place of an index.

Example:

```
s = Cards1.GetCards(SPADES)
```

See Also

Block Moves

Querying the Stack

♥♦♣♠

RemoveCards

Query the deck for all the cards matching certain criteria. Returns a String containing the card binary data and removes the matching cards from the stack.

Like all queries, it uses a syntax similar to accessing a property array, but takes a query pattern in place of an index.

Example:

```
Cards2.AddCards = Cards1.RemoveCards(ACE)
```

See Also

Block Moves

Querying the Stack

♥♦♣♠

Sorting

Determine the sorting criterion to be used when triggering a CARDS_ACTION_SORT **Action**.

The following table lists all the available options, according to the constants defined in CARDPACK.BAS:

Constant	Value	Meaning
CARDS_SORT_BYSUIT	0	Sort suit-first
CARDS_SORT_BYVAL	1	Sort value-first

By default, this property is set to 0 (CARDS_SORT_BYSUIT)

Example:

```
' sort the stack value-first
Player1.Sorting = CARDS_SORT_BYVAL
Player1.Action = CARDS_ACTION_SORT
```

See Also

Actions.

♥♦♣♠

SpreadDir

This attribute defines the 'perspective' to use when the cards are vertically stacked, or the 'direction' in which the cards are spread when the stack is strewn across the table. The direction can be (relative to the screen position of card 0) down, up, left, right or any diagonal combination.

The following table lists all the available directions, according to the constants defined in CARDPACK.BAS:

Constant	Value	Meaning
CARDS_SPREAD_UP	0	
CARDS_SPREAD_DOWN	1	
CARDS_SPREAD_RIGHT	2	
CARDS_SPREAD_LEFT	3	
CARDS_SPREAD_UP_RIGHT	4	
CARDS_SPREAD_UP_LEFT	5	
CARDS_SPREAD_DOWN_RIGHT	6	
CARDS_SPREAD_DOWN_LEFT	7	

By default, this property is set to 0 (CARDS_SPREAD_UP).

Note that, when using a spread direction that puts card 0 at the right or bottom of the stack, you may also want to turn off the **AutoSize** property, because, otherwise, when removing cards from the top of the stack, it will appear to 'move up' the screen. Actually what's happening here is that the control is shrinking while keeping the Top, Left co-ordinates fixed.

Example:

```
' Flush right deck across the table
Deck.SpreadDir = CARDS_SPREAD_RIGHT
Deck.SpreadStyle = CARDS_SPREAD_TIGHT

' Then pack it up again
Deck.SpreadDir = CARDS_SPREAD_UP_RIGHT
Deck.SpreadStyle = CARDS_SPREAD_SLANTED
```

See Also

Laying Out The Stack

♥♦♣♠

StackBack1

This property determines, which back design is used for the cards in the first and every other odd deck.

This property is consulted when presetting the stack with the **NumCards** property. For example, when you preset the stack to 108 cards, cards 0..53 (the first pack) will have the back specified by this property, while cards 54..107 will have the back specified by the **StackBack2** property.

You can, however, use the property to apply a global change to the back design of all cards in a stack that matched the *previous* setting. Suppose, for example, that cards 0..51 were set to **Deck** 1 and **Back** design #3 through this property when the stack was created. Changing StackBack1 from 3 to 5 will affect all and only the cards that previously had 3 as their back design. Changing StackBack2 will only affect the *other* cards in the stack

The value you assign this property to represents the *index* of the card back bitmap from the DLL. In the standard CARDS.DLL, back bitmap # 1 is located at resource # 53. You chose this by selecting StackBack1 = 1.

By default, this property is set to 1.

Example:

```
' Make fancy double-decker
Fancy.StackBack1 = 3
Fancy.StackBack2 = 5
Fancy.NumCards = 108
```

See Also

Laying Out The Stack
Presetting The Stack

♥♦♣♠

StackBack2

This property determines, which back design is used for the cards in the second and every other even deck.

This property is consulted when presetting the stack with the **NumCards** property. For example, when you preset the stack to 108 cards, cards 54..107 (the second pack) will have the back specified by this property, while cards 0..53 will have the back specified by the **StackBack1** property.

You can, however, use the property to apply a global change to the back design of all cards in a stack that matched the *previous* setting. Suppose, for example, that cards 54..107 were to **Deck** 2 and **Back** design #5 through this property when the stack was created. Changing StackBack2 from 5 to 3 will affect all and only the cards that previously had 5 as their back design. Changing StackBack1 will only affect the *other* cards in the stack

The value you assign this property to represents the *index* of the card back bitmap from the DLL. In the standard CARDS.DLL, back bitmap # 2 is located at resource # 54. You chose this by selecting StackBack1 = 2.

By default, this property is set to 2.

Example:

```
' Make fancy double-decker
Fancy.StackBack1 = 3
Fancy.StackBack2 = 5
Fancy.NumCards = 108
```

See Also

Laying Out The Stack
Presetting The Stack

♥♦♣♠

DrawStepHoriz

Indicates the horizontal distance between two consecutive cards in the stack Usually this property is read-only and is computed according to the current setting of **SpreadDir** and **SpreadStyle**. It can only be written to when **SpreadStyle** is set to 'User-Defined', in order to lay out the stack manually.

Example:

```
' Draw cards from lower left to upper right corner,
' but just barely spreading up.
Cards1.NumCards = 4
Cards1.AutoSize = True
Cards1.SpreadStyle = CARDS_SPREAD_USERDEF
Cards1.DrawStartX = 0
Cards1.DrawStartY = 15
Cards1.DrawStepHoriz = 17
Cards1.DrawStepVert = -5
```

See Also

Laying Out The Stack

♥♦♣♠

DrawStepVert

Indicates the vertical distance between two consecutive cards in the stack. Usually this property is read-only and is computed according to the current setting of **SpreadDir** and **SpreadStyle**. It can only be written to when **SpreadStyle** is set to 'User-Defined', in order to lay out the stack manually.

Example:

```
' Draw cards from lower left to upper right corner,
' but just barely spreading up.
Cards1.NumCards = 4
Cards1.AutoSize = True
Cards1.SpreadStyle = CARDS_SPREAD_USERDEF
Cards1.DrawStartX = 0
Cards1.DrawStartY = 15
Cards1.DrawStepHoriz = 17
Cards1.DrawStepVert = -5
```

See Also

Laying Out The Stack

♥♦♣♠

DrawCardEvery

Indicates the 'sampling rate' of cards to be written. Usually this property is read-only and set to 1, meaning 'draw every card'. It can only be written to when **SpreadStyle** is set to 'User-Defined'. in order to lay out the stack manually. You may want to override this setting in case you are displaying a very thick deck that will probably look too high in the standard 'Slanted' SpreadStyle.

Example:

```
' Create a bulky deck
Cards1.SpreadStyle = CARDS_SPREAD_SLANTED
Cards1.NumCards= 108

' Make it look a little leaner
Cards1.SpreadStyle = CARDS_SPREAD_USERDEF
Cards1.DrawCardEvery = 2
```

See Also

Laying Out The Stack

♥♦♣♠

DrawStartX

Indicates the horizontal position of card # 0 in the stack relative to the control position. Usually this property is read-only and is computed according to the current setting of **SpreadDir** and **SpreadStyle**.. It can only be written to when **SpreadStyle** is set to 'User-Defined', in order to lay out the stack manually.

Example:

```
Cards1.SpreadStyle = CARDS_SPREAD_USERDEF
Cards1.DrawStartX = 20
Cards1.DrawStartY = 20
```

See Also

Laying Out The Stack

♥♦♣♠

DrawStartY

Indicates the vertical position of card # 0 in the stack relative to the control position. Usually this property is read-only and is computed according to the current setting of **SpreadDir** and **SpreadStyle**. It can only be written to when **SpreadStyle** is set to 'User-Defined', in order to lay out the stack manually.

Example:

```
Cards1.SpreadStyle = CARDS_SPREAD_USERDEF
Cards1.DrawStartX = 20
Cards1.DrawStartY = 20
```

See Also

Laying Out The Stack

♥♦♣♠

AutoSize

This property tells the control whether it should change its size as cards are added and deleted. It is normally good to have when adding cards, because you then never have to worry about the maximum area covered with cards, the control will take care of that.

On the other hand, this option forces the control to recalculate some layout parameters and fully redraw itself when cards are added and removed, causing a lot of shuffling that you may want to get rid of. A little experimentation will tell you if you can live with the extra refreshes or if you can efficiently remove them by setting this property to False.

By default, this property is set to True.

Note that, when using a spread direction (**SpreadDir**) that puts card 0 at the right or bottom of the stack, you may also want to turn off the AutoSize property, because, otherwise, when removing cards from the top of the stack, it will appear to 'move up' the screen. Actually what's happening here is that the control is shrinking while keeping the Top, Left co-ordinates fixed.

Example:

```
Cards1.AutoSize = False
```

See Also

Laying Out The Stack

♥♦♣♠

CardHeight

This property indicates the vertical size of a card. By default it is read-only, showing the size as read from the bitmap resource in the library. However, if you have set the **CardStretching** property to True, you can then set CardHeight to a different value and get the card bitmap scaled to the specified size.

Example:

```
'Make these cards half the size'
Cards1.CardStretching = True
Cards1.CardHeight = Cards1.CardHeight / 2
Cards1.CardWidth = Cards1.CardWidth / 2
```

See Also

Card Pictures
Laying Out The Stack

♥♦♣♠

CardWidth

This property indicates the horizontal size of a card. By default, it is read-only, showing the size as read from the bitmap resource in the library. However, if you have set the **CardStretching** property to True, you can then set CardWidth to a different value and get the card bitmap scaled to the specified size.

Example:

```
'Make these cards half the size'
Cards1.CardStretching = True
Cards1.CardHeight = Cards1.CardHeight / 2
Cards1.CardWidth = Cards1.CardWidth / 2
```

See Also

Card Pictures
Laying Out The Stack

♥♦♣♠

CardStretching

When set to True, this property 'unlocks' the **CardWidth** and **CardHeight** properties, allowing you to manually set the size of the cards, overriding the default values indicated in the resource DLL.

By default, this value is set to False.

Example:

```
' Make small copy of big deck
Small.CardStretching = True
Small.CardWidth = Big.CardWidth / 2
Small.CardHeight = Big.CardHeight / 2
Small = Big
```

See Also

Card Pictures
Laying Out The Stack

♥♦♣♠

SpreadStyle

This attribute specifies the basic stack visual organisation. You can select from 5 predefined spreading styles, or set the property to 'user defined' and finely tune the spreading effect on your own.

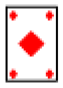
For example, setting SpreadStyle to 1 (CARDS_SPREAD_SLANTED) can be used to simulate a slight perspective view effect.

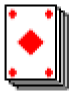
Note that the **SpreadDir** attribute can be used as an additional modifier, indicating the direction in which the stack should be spread or viewed in perspective.

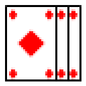
By default, this value is set to 0 (CARDS_SPREAD_STACKED).

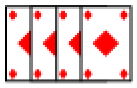
The following table lists the possible values, as defined in CARDPACK.BAS:

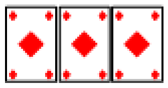
Constant	Value	Meaning
CARDS_SPREAD_STACKED	0	Vertical view from above: only top card shows
CARDS_SPREAD_SLANTED	1	Slight perspective effect: only borders visible at edge
CARDS_SPREAD_TIGHT	2	All cards recognisable from border (suitable for large hands)
CARDS_SPREAD_WIDE	3	Half cards are visible (suitable for small hands)
CARDS_SPREAD_STREWN	4	All cards fully visible side by side
CARDS_SPREAD_USERDEF	5	You can set all the positioning properties yourself


Vertical Stack


Slant Stack


Tight Spread


Wide Spread


Strewn

Example:

```
' Show all cards that player 1 has
Player1.SpreadStyle = CARDS_SPREAD_TIGHT
Player1.Action = CARDS_ACTION_TURN_UP
```

See Also

Laying Out The Stack

♥♦♣♠

Action

Action is a special property that is commonly used in VBXs to work around the problem that it is not possible to add custom methods to the objects. Each method is thus implemented as a command that is assigned to this property.

For OCX controls, the Action property is still supported, to maintain compatibility with existing VBX code, but is considered obsolete. You can use one of the **Methods** instead.

These are supported operations, as defined in CARDPACK.BAS. Their use is described in **Actions**.

Constant	Value	Meaning
CARDS_ACTION_NONE	0	No operation
CARDS_ACTION_SHUFFLE	1	Shuffle the stack
CARDS_ACTION_SORT	2	Sort the stack (suit order criterion)
CARDS_ACTION_TURN_UP	3	Set all cards facing UP
CARDS_ACTION_TURN_DOWN	4	Set all cards facing DOWN
CARDS_ACTION_DESELECT	5	Marks all cards as deselected
CARDS_ACTION_SELECT	6	Marks all cards as selected
CARDS_ACTION_PACK	7	Pack the stack, removing empty slots
CARDS_ACTION_CLEAR	8	Clear the stack, removing all cards
CARDS_ACTION_RESET	9	Reset stack just as when created

Note that all actions also trigger a **Change** event on the control.

Example:

```
' shuffle stack
Stack.Action = CARDS_ACTION_SHUFFLE
```

See Also

Sorting

Methods

PackOnRemove

♥♦♣♠

Events

The control supports the events listed in the following table:

Change	*GotFocus	*LinkError	*MouseMove
*Click	*KeyDown	*LinkNotify	*MouseUp
*DblClick	*KeyPress	*LinkOpen	
*DragDrop	*KeyUp	*LostFocus	
*DragOver	*LinkClose	*MouseDown	

Events marked with an asterisk (*) are standard Visual Basic events.

Change

The control overrides the standard behaviour of this event, which is usually fired by VB when the control contents change due to an update through a DDE link. In our control this event is triggered whenever a contents change occurs that may be relevant to the game.

The following events will trigger *Change*:

Card value, suit, etc. modifications
Cards being turned up and down
Adding And Removing Cards
Block Moves
All **Actions**

Changing the selection status of individual or group of cards will *not* trigger Change, because selection happens prior to a game move and may change several times before cards are played.

Block moves, queries and action that affect groups of card will only trigger *one* Change event. They are therefore much more efficient and faster than their counterparts that operate on individual cards.

The control automatically protects you from triggering a recursion inside the Change event. This happens, for example, when handling the cards that were played and put down on the table:

```
Sub OnTable_Change ( )
        If OnTable.NumCards = 2 Then
                w% = CheckWinner
                Player(w%).AddCards OnTable.Cards
                OnTable.NumCards = 0      ' in theory could fire a Change again, but doesn’t
```

See Also

Manipulating Cards
Block Moves
Adding And Removing Cards
Querying The Stack

♥♦♣♠

Standard Properties

Please refer to the Visual Basic Custom Controls on-line help documentation for information on this property, event or method.

If you are using Visual Basic 4, the information you are looking for may be included in the Custom Controls Reference Help File: **CTRLREF.HLP**

♥♦♣♠

CardsLibrary

This property allows you to explicitely specify a full pathname to the cards DLL to be used to provide the pictures for a specific control instance.

When left unspecified or set to an empty string (the default setting), this property will cause the control to attempt loading card pictures from the standard CARDS.DLL that should be found in every Windows games installation.

In case the specified DLL cannot be loaded, the control will look for one of the standard CARDS.DLL on the system.

Note:

This property is only available in OCX versions of the control.

See Also

Card Pictures
Installation
CARDS.DLL

♥♦♣♠

CurrentCard

This read-only property removes from the stack and returns the card that was under the mouse when the most recent event was detected, regardless of its selection status. This property provides a quick shortcut when you have to pick cards from the player’s hand without waiting for selections.

If you need to check the card before removing it, you may use the **Current** or the **LastSelected** properties.

This property will return CARD_EMPTY if no card is under the cursor.

This property is only available in the OCX version of the control.

Example:

```
' on mouse click, drop card on table
Table.TopCard = Player1.CurrentCard
```

See Also

Card Selection

♥♦♣♠

TopCards

This read-only property extracts and returns all the cards in a stack from the indicated index to the top of the stack. It can be use to very quickly split a deck and move part of it to another one without having to loop on every card.

Do not confuse this property with **TopCard**, which only retrieves 1 card.

This property is only available in the OCX version of the control.

Example:

```
‘ in solitaire, move part of a column to another one.
Column(2).AddCards = Column(3).TopCards(5)
```

See Also

Adding And Removing Cards

♥♦♣♠

BottomCards

This read-only property extracts and returns all the cards in a stack from index 0 up to the indicated index. It can be use to very quickly split a deck and move part of it to another one without having to loop on every card.

Do not confuse this property with **BottomCard**, which only retrieves 1 card.

This property is only available in the OCX version of the control.

Example:

```
‘ swap top and bottom halves of the stack
Deck.AddCards = Deck.BottomCards(20)
```

See Also

Adding And Removing Cards

♥♦♣♠

DrawCard

This graphical method allows you to quickly draw individual cards on any form or picture control. It is intended for creating simple animations, as shown in the *About Box* of the sample game included in the distribution archive, *Briscola*.

Syntax:

cardpack.**DrawCard** *hDC, descr%, xPos%, yPos%, width%, height%*

You will have to pass this method the *device context* of the target form or control. In Visual Basic, this is represented by the **hDC** property. Note that, when drawing onto a form, your picture will go *under* any control on the form. If you want to draw *over* anything that is on the form, you will have to turn off the form's **ClipControls** property.

When you are finished with your animation, you may want to "clean up" the form from your scribbling. You can do that by calling the form's **Refresh** method.

You can indicate a custom size (width and height) for the card to be drawn. Take advantage of this to create some nice animation effects. These parameters are optional, by default the card will be drawn in its natural size.

Example:

```
' draw card with increasing height, centered on axis, at x_pos%, y_pos%
' assume cpk is some Cardpack control
' see sample programs for more complex examples

For h% = 0 To cpk.CardHeight
    y% = y_pos% + (cpk.CardHeight - h%) \ 2
    cpk.DrawCard Me.hDC, cpk.Card(0), x_pos%, y%, cpk.CardWidth, h%
Next h%
```

See Also

Methods
Samples
Dragging Cards

♥♦♣♠

Web Pages

Browser support

The Cardpack control (32-bit OCX version), like any ActiveX control, can be inserted in a Web page. You can even write a card game in VBScript or JavaScript and show it over the Web from your site.

However, remember that, for the time being, the only Browser that directly supports ActiveX controls is Microsoft Internet Explorer. The Netscape browsers require a special plug-in (available at **http://www.ncompasslabs.com**) before they can access pages containing ActiveX components. Even with this plug-in, the Web pages still need to be prepared in a special way and you will still run into some subtle incompatibilities in details of control layout, resizing and refreshing. Additionally, this plug-in currently does not support the Microsoft licensing mechanism, so your web pages will always generate license warnings from Netscape.

Another important point is that the control is not *signed* yet, so Internet Explorer may show a security warning when downloading it. In order for the control to be accepted by Internet Explorer, the user of your web pages may have to tweak the default safety settings:

IE 3.x	Set Safety Level to medium or low in the **View->Options->Security** panel. Also make sure that, on the same panel, the options in Active Content are enabled.
IE 4.x	Select **View->Internet Options->Security**. Set the level to low or custom. Make sure that all options relating to ActiveX download and usage are set to either *prompt* or *enable*.

Creating Web Pages

If you want to create web pages including ActiveX controls, the easiest way to do it is to use a package like Microsoft Front Page and follow the included instructions.

Just in case you need to edit the HTML pages by hand, the following is a quick overview explaining how to include the Cardpack ActiveX control in the page and drive it with some VBScript code. We will assume that you already know the basics of Web technology and HTML.

1. Embedding the Cardpack control in the HTML page:

To add the object to an HTML page, use the <object>...</object> tag, as shown in the following example:

```
<object id="Cards1"
        name="Cards1"
        classid="CLSID:6B17E7E0-AECA-11D0-B4D8-444553540000">

<param name="NumCards" value="8">
<param .....>

        Cardpack Control 1
</object>
```

Notes:

The "name" qualifier is the control name you will use to reference this particular instance of the control from the script procedures.

The value of the "classid" qualifier must be entered exactly as shown above.

The "Cardpack Control 1" line is an arbitrary text string which is shown in place of the control, in case the browser cannot display it.

2. Presetting the control properties

Use the <param> tag within the <object></object> tags. Any control property can be specified here, along with the value. You can preset none, one, or any number of properties.

```
<param name="NumCards" value="8">
```

3. Getting the control to download and install

Use the optional "codebase" part of the <object> tag, as shown in the following example:

```
<object id="Cards1"
        name="Cards1"
        classid="CLSID:6B17E7E0-AECA-11D0-B4D8-444553540000"
        CODEBASE="http://zz.ww.com/dir/cardpk32.ocx">
        <param ...><param ...>

        Cardpack Control 1
</object>
```

4. Licensing your Web Pages

All programs written using the Cardpack control require a license to execute. In the case of a Web-page, the license must reside on the server, in a special file known as a "license package" (.LPK). Internet Explorer will fetch it and ask the controls to verify that the license is indeed valid. This is the general way of using licensed ActiveX controls on the Web, so it applies to all licensed controls you want to use, not just Cardpack.

Supposing that you have the ordinary license (.LIC) file for the Cardpack control, you will still need to create the license package file (.LPK) yourself. The procedure is rather complicated, but is fully explained at **http://support.microsoft.com/support/kb/articles/Q159/9/23.asp**.

In a nutshell, you need to:

1. Download the ActiveX SDK archive from the Microsoft Web site (**http://www.microsoft.com/activex**) and locate the tool called **Lpk_tool.exe**.
1. Run this tool on a machine that has the Cardpack control properly installed, along with the CARDPACK.LIC file.
1. You will be able to select a number of controls from those installed on that computer. Make sure the Cardpack control is included, and tell the tool to generate the .LPK file.
1. Include on your Web page exactly one instance of a special ActiveX control, called "License Manager", using the following lines exactly as shown:
   ```
   <OBJECT CLASSID = "clsid:5220cb21-c88d-11cf-b347-00aa00a28331">
   <PARAM NAME="LPKPath" VALUE="relative URL to .LPK file"> </OBJECT>
   ```
1. Include any instance of the Cardpack control after this object.

As mentioned before, unfortunately the Microsoft ActiveX Internet licensing scheme will not currently work with Netscape, even when the Ncompass plug-in is installed.

5. Driving the control

To perform actions on the control from a "program" written in VBscript, use the <script> tag in the body of the page. You can insert VBScript code there, using most of the features of Visual Basic. The syntax for accessing properties and methods of the control is also alike, as shown in the following fragment (it assumes you have a Cardpack control instance named "Cards1" on the page):

```
<body>
<script language="VBScript">
<!--
    Sub window_onLoad()
        Cards1.Turn 1 'flip the cards after page is shown
    end sub
--> </script>
```

To bind a form button to an action on the control, proceed as in the following example:

```
<input type="button" name="BFlipUp"
       value="Flip Up!" language="VBScript"
       onclick="Cards1.Turn 1">
```

6. Firing script procedures from control events

Just use the normal Visual Basic syntax within your <script>, as shown in the following example:

```
Sub Cards1_Click
  ' your code here
end sub
```

Internet Explorer will execute the code whenever the instance of the control called "Cards1" is clicked.

♥♦♣♠

CARDS.DLL

This control can use the card bitmaps it finds in SOL.EXE or CARDS.DLL. These are components of the *Solitaire* and other card games which are provided in different versions of MS-Windows and in the Windows Entertainment Packs. Since these are Microsoft products, we cannot distribute them with the control. So, first of all, make sure you have installed the card games that came with your system.

Unfortunately, 16-bit versions of the control require a 16-bit version of CARDS.DLL, while 32-bit versions of the control require a 32-bit version of CARDS.DLL. You need to be aware of this, because with some set-ups you may be missing an appropriate version of the library on the system.

Apparently Windows95 and Windows 98 installations only come with 16-bit versions of card games and CARDS.DLL (exactly like Windows 3.1). These will work fine with either the VBX version or with the 16-bit OCX version of the control, but *not* with 32-bit versions of the OCX control.

On the other hand, the version of CARDS.DLL installed in the System32 directory on Windows NT systems is a true 32-bit DLL and can be used with the 32-bit OCX version, but *not* with either of the 16-bit versions.

To sum things up, this table shows you the combinations of control and operating system that will work right away, that can be made to work or that will never work:

	Windows 3.X	**Windows 95/98**	**Windows NT**
16-Bit VBX	OK	OK	Needs 16-Bit DLL (1)
16-Bit OCX	OK	OK	Needs 16-Bit DLL (1)
32-Bit OCX	NO	Needs 32-Bit DLL (2)	OK

(1) **16-bit** versions of the control on **Windows NT** require you to get a copy of a 16-bit CARDS.DLL from a Windows 3.1 or Windows 95/98 system and copy it to the C:\WINNT\SYSTEM directory on the NT machine. You *may* already have a copy installed if the system was upgraded from a previous Windows 3.x or Windows 95 installation.

(2) **32-bit** versions of the control on **Windows 95/98** require you to get a copy of a 32-bit CARDS.DLL from the C:\WINNT\SYSTEM32 directory on an NT computer and copy it on your W95 computer in C:\WINDOWS\SYTEM as **CARDS32.DLL**

OK, so you are missing the right DLL and cannot get it? There is one for you to use with the control in the CARDSDLL **Sample**. Note that these sample DLLs do not provide the code needed by Solitaire and other standard Window games. They will only work with the Cardpack control, so make sure you don't overwrite one of the original CARDS.DLL you have on your system.

Please also note that, in the last case above, changing the name of the library is necessary in order to avoid deleting the existing 16-bit CARDS.DLL that will reside in the same directory.

Finally, the 32-bit version of the control has *not* been tested on old Win32s systems. It is *not* known if it will behave properly on such machines (if any are still around).

Of course, most of the notes above do not apply if you plan on providing your own **CardsLibrary**. In that case the only limitation you will have is running the 32-bit version of the control on Windows 3.1.

♥♦♣♠

Samples

The following samples are normally contained in the distribution package(s) or can be requested from the author. If you used the **SETUP.EXE** program to install the package, the samples should be found under **"C:\Program Files\Cardpack"** or **"C:\Cardpack"**.

BRISCOLA	A complete Visual Basic program based on a popular Italian card game. The rules of the game are easy enough that most of the code is used to implement some interesting sample functionality, such as playing against the computer, auto-play and even networking using NetDDE.
CARDSDLL	A very simple example that shows how to use a custom cards DLL. A sample cards DLL is provided, along with a Visual Basic test programs. **Note:** You can always use these DLLs with the control in case you are missing one on your system. See **CARDS.DLL** for details.
WEBTEST	A simple example showing how to embed the Cardpack control in a Web page and drive it using some VBScript code. The web page should be loaded from Microsoft Internet Explorer 3 or better. Only limited support for Netscape is provided.
WEBGAME	A full-fledged Solitaire game implemented on a Web page in VBScript. Use Microsoft Internet Explorer 3 to see this. Only limited support for Netscape is provided.

♥♦♣♠

Creating a Card Library

The Library and the Card Resources

Creating a new card picture library will require some technical expertise. First of all, you will probably have to use a “professional” Windows programming environment, like Microsoft Visual C++, that comes with a *resource editor* and a *resource compiler*.

A *Resource Editor* is a program that lets you create graphical objects used by Windows applications, like icons, small bitmaps, dialog boxes, menus, cursors, etc. A *Resource Compiler* is a program that takes all these elements and links them into an executable program or into a *Dynamic Link Library* (DLL).

With most of these tools, creating a new library may also require writing some lines of code. With other tools, such as the Borland *Resource Workshop*, you may get by without any coding.

As happens with the original CARDS.DLL from Microsoft, you also have to package the card pictures in a DLL. For compatibility with the Cardpack control, your new library should respect the resource ID# mapping of the original CARDS.DLL for all standard cards. This will let the control find the appropriate picture for a card:

Picture	**Resource #**
Clubs	01..13
Diamonds	14..26
Hearts	27..39
Spades	40..52
Backs	53..66
Empty	67..68

Optionally, one or two Joker bitmaps can be included as resource #70 and #71(these IDs are free in the original CARDS.DLL provided with Windows).

If resources #70 and #71 are unavailable, as happens with the original CARDS.DLL, the control will use one of the card back designs (currently #56, the robot) as a Joker.

Graphics and Colour

By default, the control will adapt itself to the size of the card bitmaps that it finds in the library. The card pictures should all have the same width and height. At run-time, the Cardpack control lets you accept this default size, or stretch the images to different sizes.

At the moment, the control does not support *masking*, so it is not possible to design cards with real rounded edges, where the background shows through under the corners.

The control supports loading and using cards with palettes containing any number of colours. In particular, it is possible to create cards pictures with 256 colours. The control will notice if the display supports the use of palettes (i.e. not True Colour). It will then look for a palette in the ‘test’ card it loads from the library (the first back design, resource #53), at the same time it is recording the default size.

That palette is recorded and used for *all* the cards. For this reason, all cards in excess of 16 colours should be prepared using the same palette. You can do this with a good graphic editor like *Paint Shop Pro*, for example. If possible, make sure that the Windows’ standard colours are also included in the palette. This will make colour mapping easier.

It is also possible to mix 256-color pictures (e.g. the card backs) with standard low-colour pictures (the card faces). Just make sure that the “test card” mentioned above (resource #53) is one of those with the palette.

Finally, remember that the higher the number of colours, the more memory it will take to store and load the card pictures. Do not use schemes with many colours unless you are really producing an artistic card set. Even in that case, it is probably a good idea to make sure the number of colours is reduced to the minimum it takes to still have a good effect.

Important Notes

Installing in your system a new CARDS.DLL which is incompatible with the original one from Microsoft will break your Windows games, such as Solitaire. We strongly suggest you pick a different name for your DLL and then explicitly tell the control to load that library.

If you are compiling your DLL from some programming environment, we also strongly suggest you pick a module name (the *internal* DLL name you specify in the .DEF file) which matches your original name and does not conflict with CARDS.DLL (e.g. pick module name "mycards" and file name "mycards.dll"). If you don't, you may encounter some very strange behaviour in the Windows games or in your own programs.

The reason is that Windows looks up DLL's by their *file* name, but then can only tell them from one another by their *internal* name. If you have mismatches or duplicate names, Windows will get very confused as to which DLL it should load and use.

In the distribution archive we provide a custom cards DLL for you to try out, as part of the CARDSLL **sample**.

♥♦♣♠

Control Variants

The Cardpack control comes in three different variants, as described below. Depending on your development environment, you may want to install any or all of these.

CARDPACK.VBX	*(available in the VBX distribution only, until version 1.1)* is suitable for use with 16-bit versions of Visual Basic. It will work fine with Visual Basic 3.0 and 4.0/16 bit. It will *not* work with 32-bit versions of the language or with Visual Basic for Applications.
CARDPK16.OCX	*(available in the OCX distribution only, until version 1.6)* is an ActiveX control suitable for use with all 16-bit environments that support this format, including Visual Basic 4.0/16 and 16-bit versions of Visual Basic for Application (i.e. 16-bit Microsoft Office products).
CARDPK32.OCX	*(available in the OCX distribution only)* is an ActiveX control suitable for use with all 32-bit environments that support this format, including Visual Basic 4.0/32 or better, 32-bit versions of Visual Basic for Application (i.e. 32-bit Microsoft Office products), Visual C++ and Web pages.

Note: "ActiveX control" is just the most recent denomination of what used to be called OLE controls, or OCX's. With respect to the functionality of this control and its documentation, all three denominations are practically equivalent.

Note: the distribution archive should also include the following essential files:

CARDPACK.HLP	*the help file*
CARDPACK.BAS	*support file for program development*

♥♦♣♠

Cardpack Control Registration

The Cardpack control is an ActiveX component for use with the Microsoft Windows family of operating systems. It was developed and copyrighted by Andy Zanna

This software is not a commercial product. It is developed, mantained and supported solely on a best-effort basis in the author's spare time. For this reason, it is not "sold", but rather distributed as *Shareware*.

The author encourages you to register the control and support its development with a contribution. The suggested minimum contribution is US$25 or equivalent in major currencies. Higher contributions, especially from commercial developers, are of course welcome.

The controls contained in the distribution archives are fully functional. However, the software contains a license mechanism intended to enforce registration before any commercial use can be made of it.

Why Register?

When you register the control, you will have the following advantages:

* You will get rid of the warning dialog boxes that normally appear every time the control is loaded;
* You will then be able to compile stand-alone applications and distribute them free of royalty;
* You will also receive notifications of updates and a right to use all minor updates (e.g. 1.x) with the same license;
* The author will give more attention to your problems, compatibly with his schedule.

Please check out the **Information** page for the detailed usage and distribution terms.

How to Register

Your registration contribution should be sent by personal check or mail money order to the name and address below.

Name:	Andy Zanna
Address:	Hauptstrasse 41 D-71263 Weil der Stadt Germany
Phone:	+49-7033-33800
E-mail:	zanna@multimedia.it

You should add another 5$ if you require that the software and the license be sent to you via ordinary mail, rather than E-mail.

You should also send a registration notice by ordinary mail or E-mail (preferably) to the address above, indicating your full name, company (if applicable), postal address, E-mail address.

♥♦♣♠