Cardbox: macros and programming.

Cardbox: macros and programming

Martin Kochanski

and the Cardbox team

Cardbox Software Limited

First published in 2004 by Cardbox Software Limited.

ISBN: 0-9548226-1-7

A catalogue record for this book is available from the British Library.

Cardbox Software Limited
Scriventon House
Speldhurst
Kent TN3 0TU
England

Preface

Macros let you automate Cardbox. At their simplest, they record your actions so that you can play them back later; but the same mechanisms that let you do this will also let you program Cardbox in increasingly sophisticated ways. Cardbox macros can control many other programs in your Windows system, which makes it easy to do things such as send emails or faxes. On the other hand, if you are familiar with programming something like Microsoft Office then you can use Office macros to drive Cardbox and thus integrate Cardbox into your Office workflows.

Most people will only need to read a small part of this book. This is true of most computer books but it's especially true of this one, so here is a quick summary of what you can expect:

Part One tells you how to record macros and play them back. You will learn how to attach macros to keystrokes, toolbar buttons, and menu items.

Part Two gives some examples of how you can add intelligence to your macros, with simple calculations or just by inserting the date into a search command (asking your diary database "what do I have to do today?"). If you're not interested in expanding your macros like this, you can stop reading at this point.

Part Three is a summary of VBScript, the programming language in which Cardbox macros are written. This is a long chapter (over 30 pages) and it's very dense, so you may find it best to skim through it on a first reading and then use it as a reference source when reading the examples that we give in the rest of the book. Even though it's long and detailed, it's still only a summary: to take your study of VBScript further, we give a list of suggested books on page 55.

Part Four introduces you to Cardbox objects. These bypass the normal Cardbox menu commands and screen display, so that you can perform searches and access records and fields faster and more directly.

Part Five describes some of the other objects that Cardbox macros can access: to read and write files, to send faxes and emails, and to drive Microsoft Word and other programs.

Part Six gives annotated examples of Cardbox macros that use objects. We explain each one in detail so that you can use the same principles in your own macros.

Part Seven looks at Cardbox from the outside: it describes how programs written in Visual Basic, VBA (the macro language used by Microsoft Office) and other languages can connect to Cardbox, control it, and use it to retrieve and update data.

Finally, **Part Eight** is a comprehensive reference list of every Cardbox object with all its methods and properties.

The history of Cardbox programming

Cardbox users have always been enthusiasts and they have never been shy about asking us for the features they want. We appreciate this because it helps us to make Cardbox better.

In 1988, though, we were worried. Some people were asking us to add simple arithmetic to Cardbox-Plus, but we saw that to add arithmetic uniformly and consistently would change the product out of all recognition and make it more complex for everyone, simply to benefit the few users who wanted the feature; and yet we could see that those who wanted arithmetic (and other extra features) had a point. The dilemma was resolved when one of us invented coprogramming.

Co-programming (pedants will insist on the hyphen) was very simple. While Cardbox-Plus was still running, the user would start another program. That other program would “type” keystrokes into Cardbox-Plus and would be able to “read” the Cardbox-Plus screen. The stroke of genius was that this would not require any special new programming interface: Cardbox-Plus simply insinuated itself into the MSDOS operating system so that a program that “printed” to one specially named file would effectively be typing into Cardbox-Plus, and if it “read” from another specially named file it would receive the contents of the Cardbox–Plus screen display. Any programming language can read and write files, so any programming language could be used to program Cardbox-Plus.

We were amazed by the success of this simple concept. Soon all manner of unexpected uses for Cardbox-Plus surfaced: a macro-processing coprogram was written; one enterprising Norwegian created a whole invoicing and stock control system in Cardbox-Plus; and to this day we use some of our own Basic coprograms to run our company accounts: there has never been any need to change them.

With the arrival of Windows this simple, universal interface was no longer possible: even “reading the screen” made little sense once screens were made of pixels rather than characters. At last, in 1992, Microsoft released OLE 2.0, a raft of technologies of which one, OLE Automation (its name has changed since then) allowed one program to drive another and extract data from it.

The initial releases of OLE Automation were unreliable and dreadfully slow, but with time and the arrival of 32-bit versions of Windows, the speed and quality have both become very good. All high-level languages and many applications now support Automation and although the promised abundance of scripting languages never materialised, VBScript has become a solid and reliable language with many powerful features. In Cardbox 3.0, we have discarded the proprietary macro language that we used in previous versions, and Cardbox macros are now written in VBScript. As you will see, the result goes far beyond the mere recording and playing back of commands, and we look forward to being amazed once again at the sophisticated things that our users achieve when they program Cardbox.

Part One
Macros

What macros are; macro facilities in different versions of Cardbox; recording macros; playing macros; managing macros; keystrokes, buttons, and menu items.

What is a macro?

A macro is a sequence of Cardbox commands that you can execute with a single action. We call this "playing" the macro.

- If you use a particular set of commands very often, a macro can save you time because you can play it in one single action.
- If you have a very specific set of commands that you use rarely (such as a monthly mailing) then storing them as a macro can save you a lot of effort in trying to remember exactly how you did things last time.

There are many ways of playing macros:

Tools > Play
F5

- The Play button on the toolbar, and the menu command Tools > Play.
- A keystroke of your choice.
- Toolbar buttons that you design yourself.
- Pushbuttons that you add to a format design.
- Items that you add to the Special menu.
- Macros that start automatically when you open Cardbox.

There is no limit to the number of macros you can have, and you can organise them in groups so that you can find them easily.

Adding intelligence to macros

Internally, macros are mini-programs written in Microsoft's VBScript programming language. When you tell Cardbox to record your actions and turn them into a macro, Cardbox actually creates a series of commands in VBScript.

This fact is interesting because it means that you can add intelligence to your macros. You can make simple changes – for example, you could record a search command and then modify it so that it always searched records marked with today's date – or you can write elaborate macros in VBScript using loops, conditional actions, calculations, and all the other features of the programming language.

VBScript can connect to and communicate with anything that supports the ActiveX or OLE Automation standard, and this includes all the programs in Microsoft Office as well as many of the built-in features of Windows. So you could write a Cardbox macro that takes a selection of records representing business contacts, formats a standard letter to each of them, and tells Windows to send each of those standard letters as a fax.

Using other programming languages with Cardbox

Any scripting or programming language that supports Automation or ActiveX can be used to drive Cardbox, using the same methods that VBScript macros use. One particularly interesting example is VBA ("Visual Basic for Applications"), which is used by Microsoft Office as its macro language. Just as VBScript macros in Cardbox can send commands to programs outside Cardbox, so VBA macros in (for example) Word or Excel can send commands to programs outside Microsoft Office, and so they can send commands to Cardbox as well. If you're familiar with writing macros in Microsoft Office, it is not difficult to extending them to communicate with Cardbox.

Macro facilities in Cardbox

Professional Edition

The Professional Edition can record, edit and manage macros, and the rest of this chapter tells you how to do this.

Client and Home Editions

The Client and Home Editions of Cardbox don't have facilities for recording, editing, or managing macros, but they can play them. This means that someone who uses the Professional Edition to set up a database can then make it available to people without the Professional Edition, and all the macros will still work.

Recording and saving macros

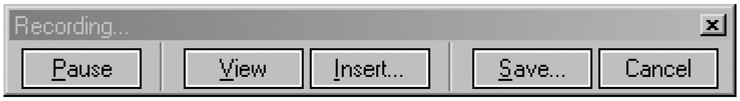
Recording

Tools > Record

The simplest way of creating a macro is to record it, and the simplest way of starting to record a macro is to press the Record button. It will start flashing to show that your actions are being recorded.

When you start recording a macro, the Record button starts to flash and the Recording button bar appears: this gives you easy access to the commands that control the recording process.

The button bar floats on top of Cardbox: if it covers up something important, you can drag it out of the way with the mouse.



The same commands that are listed in the button bar can also be reached by pressing the flashing Record button or by opening the Tools > Recording menu.

While you are recording a macro, every Cardbox command that you use will be stored in the macro. There are very few commands that can't be recorded, and Cardbox will disable those so that you can't use them accidentally while recording.

Inserting special actions

Insert...

> Insert

While you are recording a macro, you can insert special actions into it that affect the way the macro behaves when it's played back.

Message (Popup) displays a message and waits for the user to acknowledge it before continuing with the macro. This action can also give the user the chance to cancel the macro.

Help Point 555 describes all the options associated with these actions and explains what you can use each of them for.

Status Bar Message displays a message on the status bar and continues with the macro. The message remains until another Status Bar Message action replaces it with something else.

Pause With Message works like Message (Popup) but allows the user to enter Cardbox commands before continuing with the macro. For example, you could use this action to give the user the opportunity of selecting records before they are formatted and printed. If you specify a message, Cardbox will display it along with Continue and Stop buttons; if you don't, no message will appear but the Pause button in the toolbar will flash to show that the macro is paused.

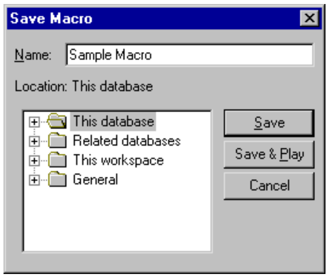
Comment In Macro Script has no effect on the way the macro works, but the comment is stored in the text of the macro. This can be a useful marker if you are planning to edit the macro by hand later on.

Saving a recorded macro

● > Save

When you have finished recording a macro, press the Save button. Cardbox will show you the commands you have recorded and ask you for a description of the macro. When you've put in a description, press Save, and the Save Macro box will appear, asking you what name you want to give to the macro and where you want to store it.

Where macros are stored



There are four places where you can put your macros, and your choice depends on what the macros do and when you want them to be accessible.

Database macros are stored with the database and they aren't visible when you're in any other database. If your macro contains commands that make sense only in the database you're currently in, make it a database macro.

Related Databases macros refer to a group of databases, not just one database. They are quite an advanced feature: see Help Point 571 for an explanation.

Workspace macros are associated with your current workspace and aren't visible when you use a different workspace.

General macros are visible whenever you use this copy of Cardbox.

To select a location, click on it and then type the name of the macro in the Name box. If you want to see what macros are already stored in a location, double-click on it and it will expand.

"Managing Macros" (p.11) tells you how to create and manage folders.

If you have a lot of macros, you can subdivide the standard locations into folders: to see the folders in a location, double-click on the location name; to select the folder that you want to save a macro into, click on the folder name in the list.

Pausing or cancelling a recording

● > Cancel

● > Pause

If you change your mind about recording a macro, or find that you've entered the wrong commands, you can cancel the recording.

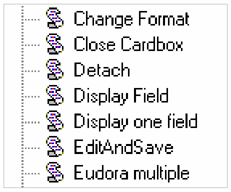
Sometimes you may want to pause a recording for a moment: during a pause, you can enter Cardbox commands but they won't be recorded. To resume recording, press the Pause button again.

Playing macros

Tools > Play
F5

The simplest way of playing a macro is to press the Play button. Cardbox will show you a list of all the available macros and you can pick the one you want. The macro list is organised as two pages, each of which gives you a different view of the macros.

The Play page



This shows you all the available macros in a single list. To find out more about a macro, click on it and Cardbox will tell you its location and description: this is why it's worth giving your macros good descriptions when you record and save them.

To play a macro, click on it and press Play; or just double-click on it.

If you want to see the text of a macro to make quite sure it is the one you want, right-click on it and select View from the pop-up menu.

The Manage page

See "Managing Macros" on page 11.

The Manage page is meant for general macro management but you can also use it to select macros and play them. Unlike the Play page, the Manage page divides your macros into separate lists according to their location.

Pausing and stopping macros

To pause a macro while it's playing, press the Pause button. You can then look at the screen and even enter Cardbox commands before pressing Pause again to continue the macro.

To stop a macro, press the Stop button on the toolbar, or press the red cross that appears on the status bar while a macro is playing.

Keystrokes, buttons, menu items

If you are going to be using a particular macro a lot, it is cumbersome to have to use Tools > Play and pick it from a list every time. You can configure Cardbox so that it can play a macro in a single action:

See page 13.

Keystrokes – you can tell Cardbox to play a macro whenever you press a certain key or combination of keys.

See page 15.

Buttons – you can add buttons to the Cardbox toolbar that activate macros when pressed. You can also design buttons into your record formats.

See page 17.

Menu items – you can define menu items that play macros: these appear as part of the Special menu in the Cardbox menu bar.

Playing macros on startup

Tools > Options » Startup

You can configure Cardbox to play a macro automatically when it starts. This can apply to a single workspace or to all workspaces.

Here are a couple of uses for startup macros:

The macro can make a standard initial selection. The sample "What must I do today?" macro (page 55) is a good example of a macro that could usefully be played automatically when a workspace is opened: if you ever don't want to start with the selection it suggests, a simple Search > Undo will get you back to viewing the whole database.

You can dedicate a workspace to a single task, such as automatically backing up a collection of databases (see page 75 for a sample macro). The macro could even close Cardbox afterwards so that everything was fully automated.

You can start with an empty workspace and use the startup macro to open a number of databases within it. We ourselves prefer to leave databases open from one session to the next, but startup macros are a legitimate alternative if you can't or don't want to do this.

This is not an exhaustive list and you may well find other uses for startup macros.

If you have a startup macro but don't want it to play on a particular occasion, just hold down the Shift key as you start Cardbox, and keep holding it down until Cardbox has finished opening. This is particularly useful if you need to adjust the contents of a workspace without the macro interfering in what you are doing.

Safety, trust, security

For more about safety levels, see Help Point 526.

A macro is essentially a small program written in VBScript, and as such it is capable of performing almost any action. This means that a malicious person could write a macro that copies your data to another database, or overwrites your files, or does something else that you wouldn't want it to do. Cardbox has features to protect you from malicious macros that are stored in remote databases.

Macros that run without restriction

General and workspace macros reside on your computer in a location that you choose. Cardbox imposes no restrictions on these macros because it assumes that you are responsible for the macros that are stored on your own computer.

In a local database – that is, in a database that you open using File > Open » My Computer – Cardbox imposes no restrictions on database or Related Databases macros, again because they live on your computer and you are responsible for them.

Macros in remote databases

Databases residing on a server are a different story. They may be databases residing on your corporate Cardbox server or they may be databases residing somewhere out on the Web. How much you trust them will depend very much on what you know about the database owner or creator.

Safety levels

Cardbox lets you define safety levels for each database you open using the Server tab. The safety level tells Cardbox how much you trust the database and controls what a macro originating from that database is allowed to do.

Untrusted is the default setting when you are connected to a server that is not on your own computer.

Untrusted –Macros loaded from an Untrusted database cannot create objects unless those objects are installed on your computer and marked as “Safe for Scripting”: in particular, the built-in Windows object that gives VBScript the ability to handle files (FileSystemObject) is not considered Safe for Scripting, which means that untrusted macros will not be able to open or create files on your computer. In addition, untrusted macros cannot access other Cardbox databases (except databases located in the same directory as the one that was the source of the macro): this is to prevent the possibility of data theft. Finally, untrusted macros cannot run programs or open external files.

Completely Untrusted – If a database is marked as Completely Untrusted then macros loaded from it can't be run at all. We have included this setting in case

Microsoft's "Safe for Scripting" mechanism ever turns out to have serious security leaks.

Trusted – Like Untrusted, except that macros loaded from this database are allowed to view or modify any open database, not just the database they have come from. They still can't create unsafe objects, open or create files, or run programs or open external files.

Highly Trusted – Like Trusted, but there is no restriction on the objects that these macros can create. This also implies access to FileSystemObject, which means that files can be created, deleted, read and written. Macros still can't run programs or open external files.

Completely Trusted is the default setting for databases stored on your own computer.

Completely Trusted – Macros loaded from this database have no restrictions

Setting up safety levels

In File > Open » Server, you can set the safety level for a particular database by picking an entry from the Safety Level list below the list of databases. You can also the set the default safety level for a whole server, or for servers in general, by pressing the Server Safety button just above the server name.

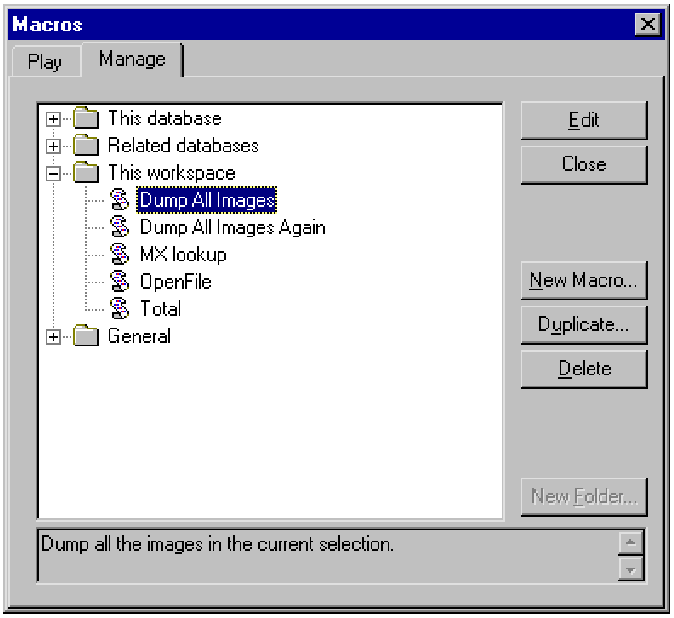
Warning: anti-virus software

To protect users of insecure email programs from the consequences of the emails they open, anti-virus software often contains a feature to restrict or block the execution of scripts, including macros written in VBScript. Older versions of anti-virus software sometimes tried to prevent scripts from being executed by any program at all, whether or not there was any worm or virus potential, and so ended up interfering with the playing of macros by Cardbox. If your anti-virus software causes this problem then you will have to reconfigure it so that it permits macros to be played: you should consult the documentation or contact the vendor for advice on how to do this. Alternatively, upgrade to a more recent version, since we have found that modern anti-virus programs seem to have a clearer idea of what kinds of scripting need to be blocked and do not block all scripts and macros indiscriminately.

Managing macros

Tools > Manage Macros

You can edit, rename and delete macros, and create macro folders, using Tools > Manage Macros. A useful shortcut to this command is to press the Play button and then select the Manage page in the window that pops up.



Dump all the images in the current selection.

At the bottom of the window you can see the description of the currently selected macro.

The macros are grouped into folders corresponding to their location (you can see a list of locations on page 6). The display works just like the folder view and Details view in Windows Explorer: you can open a folder by clicking on the + sign next to it or by double-clicking on its name.

To edit a macro, double-click on it; or select it and press Edit.

To delete a macro, select it and press Delete.

To rename a macro, right-click on it and pick Rename from the pop-up menu.

To create a brand new macro, press New Macro; or right-click on a folder and pick New Macro from the pop-up menu. You'll be taken into the macro editor, where you can type in the text of your macro.

To play a macro, double-click on it while holding down the Ctrl key; or right-click on it and then select Play from the pop-up menu.

To create a macro that is similar to an existing one, you have two choices. The traditional Windows method is to edit the macro and press Save As when you've finished editing (Cardbox will then ask you for a new name to give to the macro). The better method is to select the macro and press Duplicate, which will create a new macro that has exactly the same text as the existing one. This is safer because you can't find yourself overwriting the old macro by accident.

Managing folders

You can group your macros into folders to make your macro lists look tidier so that you can find things in them more easily. You can arrange the folders to reflect the different circumstances in which you might use the macros or the different people who might be using them.

One particularly good tip is to segregate macros that are played automatically (by keystrokes, buttons, etc) into a separate folder so that they don't confuse you when you enter the Play command manually and are looking for a macro to play.

To create a folder, right-click on the folder which is to contain the new folder and pick New Folder from the pop-up menu. The new folder will appear, with a blank name. Before you do anything else, type the name that you want the folder to have.

To rename a folder, right-click on it and pick Rename from the pop-up menu.

To delete a folder, select it and press Delete. Deleting a folder deletes all the macros and folders that it contains.

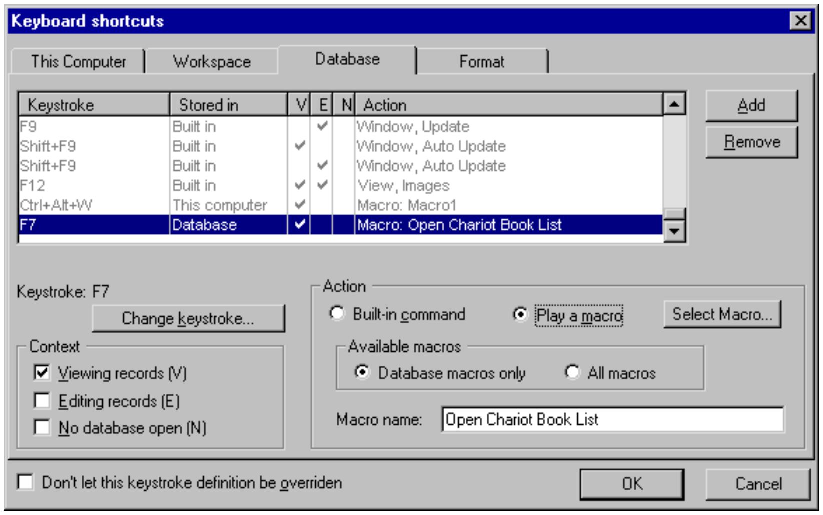
Attaching macros to keystrokes

If you use a macro often, it will save time if you can play it by pressing a single key or combination of keys: for example, F7 or Ctrl + Shift + T.

Tools > Keyboard

Tools > Keyboard lets you control all of Cardbox's keyboard shortcuts. You can reassign existing shortcuts, add new shortcuts to standard Cardbox commands, or create shortcuts that play macros.

For detailed instructions, open this command and then press the F1 key.



Keystroke	Stored in	V	E	N	Action
F9	Built in		✓		Window, Update
Shift+F9	Built in	✓			Window, Auto Update
Shift+F9	Built in		✓		Window, Auto Update
F12	Built in	✓	✓		View, Images
Ctrl+Alt+W	This computer	✓			Macro: Macro1
F7	Database	✓			Macro: Open Chariot Book List

Keystroke: F7

Change keystroke...

Context
- [✓] Viewing records (V)
- [ ] Editing records (E)
- [ ] No database open (N)

Action: Built-in command [✓ Play a macro] Select Macro...

Available macros: [✓ Database macros only] All macros

Macro name: Open Chariot Book List

[ ] Don't let this keystroke definition be overriden

OK Cancel

The properties of a keyboard shortcut

Context

The context of a shortcut says what mode Cardbox should be in for the shortcut to be available: viewing records, editing records, or with no database open. The reason for specifying the context is that many commands are available in some modes but not others. You can also use this feature to make a key do different things depending on whether or not you are editing a record, by creating two shortcuts: one for Viewing Records and one for Editing Records.

Scope

Some macros will only make sense with one particular database; others may be usable wherever you are in Cardbox. The scope of a shortcut tells Cardbox the circumstances in which the shortcut should be active.

This Computer – whenever you use Cardbox on this computer.

Workspace – only when you are using this workspace.

Database – only when you are using this database.

Format – only when you are using this particular format.

To create a shortcut with a given scope, click on the tab for that scope. The list of shortcuts includes every shortcut there is, but you can identify shortcuts belonging to the current scope because they are shown in black while the others are shown in grey.

- The Format tab is only available when you are editing a format.
- The Database tab is only available when you are editing the native format of the database.

It's possible to have shortcuts for the same key in more than one scope, and in that case the more specific scope takes precedence over the more general one: for example, if you have a Format shortcut and a This Computer shortcut, the Format shortcut will win.

Keystroke

You are allowed to assign macros to letters in the Viewing Records context, because letters on their own don't mean anything to Cardbox when you are viewing records.

In principle, most keys are usable for shortcuts – letters, numbers, symbols, and function keys – and you can modify each key with any combination of Ctrl, Alt, or Shift. There are some restrictions, though, because some keystrokes are too important to have their function changed. For example: if you assigned a macro to Alt + F then you'd have difficulty opening the File menu; if you assigned a macro to the letter A in the Editing Records context then you'd have difficulty typing the word "CAT". To avoid this sort of trouble, Cardbox prevents you from using these keystrokes.

Attaching macros to buttons

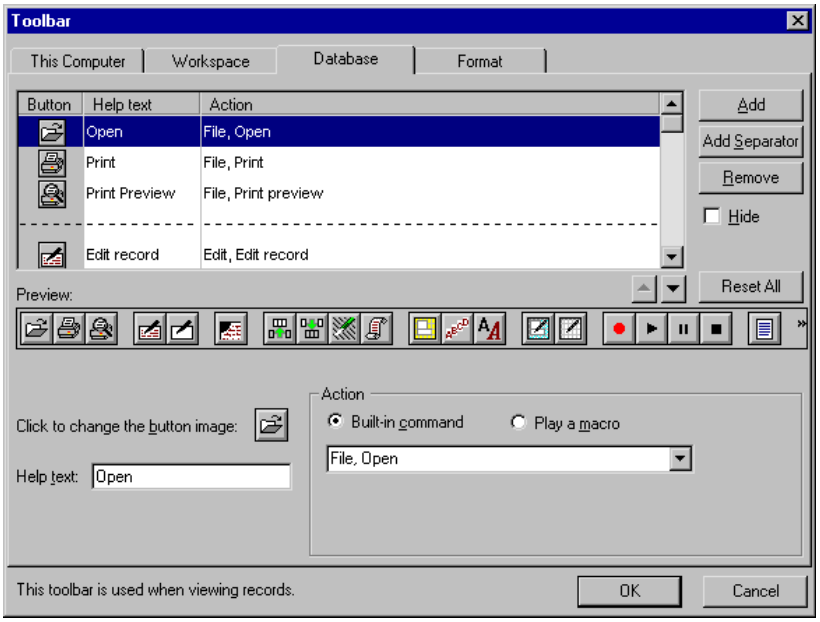
You can extend the Cardbox toolbar by adding new buttons that play macros or execute Cardbox commands. You can also add buttons to the record format itself: for details of how this works, see “Pushbuttons” in the Format Design section of the main Cardbox Book.

Tools > Toolbar

Tools > Toolbar lets you control the Cardbox toolbar. You can rearrange buttons, delete buttons, or add new buttons.

The toolbar looks different when you are editing records and when you are viewing records. To deal with this, make sure you’re in the right mode before you use Tools > Toolbar. If you want to control the toolbar that’s used when editing records, make sure you’re editing a record before you start.

For detailed instructions open this command and then press the F1 key.



Button	Help text	Action
	Open	File, Open
	Print	File, Print
	Print Preview	File, Print preview
	Edit record	Edit, Edit record

Click to change the button image:

Help text: Open

Action: [✓ Built-in command] Play a macro

File, Open

This toolbar is used when viewing records.

The layout of the toolbar

The toolbar list shows its buttons in the order in which they will appear in the toolbar: to move a button, highlight it in the list and press the up or down arrows that you'll see just below the list.

The toolbar can also contain separators, which create a small gap between one group of buttons and the next. You can create separators and move them around just like buttons.

The standard Cardbox toolbar has one row of buttons, but you can make toolbars with two rows: to do this, insert a separator and turn on its New Line property.

The properties of a toolbar button

Scope

The Scope property of a toolbar button is the same as the Scope property of a keyboard shortcut: see page 14 for details.

- You can only edit the Format scope when you are editing a format.
- You can only edit the Database scope when you are editing the native format of the database.

Image

The button image controls what the button looks like. You can give the button a solid colour or you can use one of the built-in Cardbox toolbar button designs.

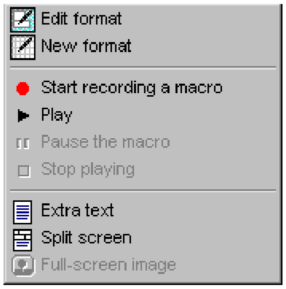
The required image size is 16 × 16 pixels.

You can also use any drawing program, such as Windows Paint, to create an image and copy it to the Clipboard. Once you've done this, click where it says "Click to change the button image" In Tools > Toolbar, and pick Paste from the pop-up menu.

Help text

The help text of a toolbar button is the text that pops up if you hold the mouse over the button for a few moments: you can use it to give the user a quick idea of what the button is for.

Cardbox also uses the help text if the window is too narrow to hold the whole of the toolbar – either because you've put very many buttons into the toolbar or because the user has made the Cardbox window very narrow. Cardbox turns the surplus buttons into a menu that pops up if you click the right-hand edge of the toolbar: the button images become menu item images and the help text becomes the text of each menu item. You can see an example on the left.



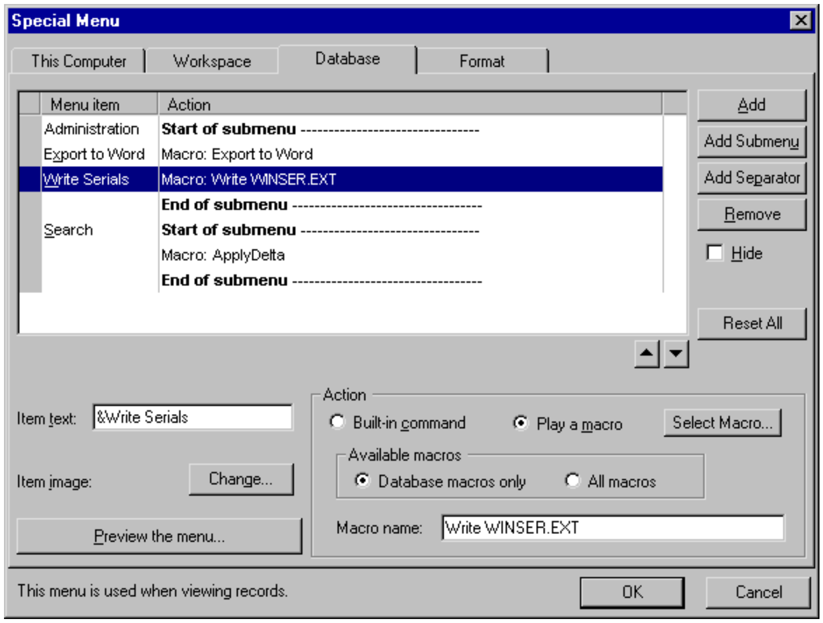
Putting macros into the Special menu

You can make macros easily accessible by adding them as menu entries to the Special menu. The Special menu comes between Tools and Window on the menu bar, but you'll only see it if it has items in it.

Tools > Special Menu

Tools > Special Menu lets you control the Special menu. You can add, edit or delete menu items, and organise them into submenus.

The Special menu looks different when you are editing records and when you are viewing records. To modify the menu that's used when you are editing records, start editing a record before you use Tools > Special Menu.



Menu item	Action
Administration	**Start of submenu** -------------------------------
Export to Word	Macro: Export to Word
Write Serials	Macro: Write WINSER.EXT
	End of submenu --------------------------------
Search	**Start of submenu** -------------------------------
	Macro: ApplyDelta
	End of submenu --------------------------------

Item text: &Write Serials

Item image:

Action: Built-in command [✓ Play a macro]

Available macros: [✓ Database macros only] All macros

Macro name: Write WINSER.EXT

This menu is used when viewing records.

The layout of the Special menu

Menu items will appear in the Special menu in the same order as they do in the Tools > Special Menu list. To move a menu item in the list, highlight it and press the up or down arrows that you'll see just below the list.

The Special menu can also contain separators: when you use the menu, these will appear as thin horizontal lines separating groups of menu items.

You can incorporate submenus into the Special menu if you like, to make the menu look simpler and to keep related menu items together. To do this, press the Add Submenu button: start and end markers will appear, and any menu item that you insert between those markers will be part of the submenu.

The properties of a menu item

Scope

The Scope property of a menu item is the same as the Scope property of a keyboard shortcut: see page 14 for details.

- You can only edit the Format scope when you are editing a format.
- You can only edit the Database scope when you are editing the native format of the database.

Image

Menu items can optionally have a small image next to them. You set up this image in the same way as for toolbar buttons: see page 16 for details.

Item text

This is the text that will appear in the menu.

Windows lets you underline one letter in the name of a menu item. That letter will then act as a keyboard shortcut. To underline a letter, type an ampersand "&" just before it in the item text.

For example, if you have an item text of `Weekly &Mailing` then this will appear in the menu as "Weekly <u>M</u>ailing", and the user can activate it with Alt + P, M (because Alt + P is the shortcut for the S<u>p</u>ecial menu).

Part Two

Intelligent macros

Tailoring recorded macros to your needs.

Tailoring a recorded macro

The basic macro mechanism built in to Cardbox lets you record your actions as a macro and repeat those actions by playing the macro. Many people never feel the need to go further than this, but it only scratches the surface of what you can do.

Internally, macros are mini-programs written in Microsoft's Visual Basic for Scripting (VBScript) programming language. When you tell Cardbox to record your actions and turn them into a macro, Cardbox actually creates a series of commands in VBScript. This means that you have all the facilities of VBScript at your disposal if you need them. Your macro doesn't have to confine itself to blindly repeating a sequence of commands.

One way of adding intelligence to a macro is to record it and then make changes to it by editing the VBScript code directly. This section shows some simple examples of what you can do by this means. Don't worry about the technical details, but try to get a feel for the idea of what macros can do once a little programming has been added.

Example: date selection

Suppose that you have a database that has a "Next Contact" or "Next Action" field with a date in it, and you want a macro that will select the records that need action today.

The date will be recorded in yyyy.mmdd format irrespective of the date format you used when performing the selection.

Turn on the macro recorder, perform the selection command, then view and edit the macro. What Cardbox has recorded will look something like this:

```
ActiveWindow.Select "NP","2004.1111"
```

You can see that the date you entered during recording has been made part of the macro. This means that if you run it tomorrow the macro will do exactly the same search as it is doing today – which is not the point. What you need to do now is make the selection command use the current date. So edit the macro as follows:

```
ActiveWindow.Select "NP",Date
```

Instead of always searching for "2004.1111", this command uses VBScript's built-in `Date` function (the value of `Date` is always today's date).

For another example, suppose that you want to do the same thing, but for dates up to and including today, rather than today only. In that case, the command you'd record would look like this:

```
ActiveWindow.Select "NP",":2004.1111"
```

and so VBScript will have to take the date and put a colon in front of it before passing it to `Select`:

```
ActiveWindow.Select "NP",":" & DateToCardbox(Date)
```

For more about DateToCardbox, what it does and why it is needed, see "Dates and Strings" on page 35.

This looks a little more complicated, because VBScript has to convert the date into a text string before inserting the colon and we have to control that conversion to make sure that it comes up with a date format that Cardbox can understand. Here, step by step, is what the macro will do when it is played:

1. Get today's date (`Date`).
2. Convert it into Cardbox format (`DateToCardbox`).
3. Put a colon in front of it (`":" &`).
4. Pass it to Cardbox as part of a selection command.

Once you've got the hang of this, you can construct endless variants. Here, for example, we're selecting records with a date within the coming week:

```
ActiveWindow.Select "NP",DateToCardbox(Date) & “:” _
                              & DateToCardbox(Date+7)
```

If you run this macro on 28 December 2004, the dates selected will be from 28 December 2004 to 4 January 2005, inclusive.

Example: tax calculation

If you are using Cardbox for your accounts then you will come across VAT (Value Added Tax). Most ledger applications have three fields, for the net (tax-exclusive) amount, the tax, and the gross (tax-inclusive) amount: let's call them NET, VAT, and TOTAL. This situation provides scope for writing some useful macros.

While entering receipts for cash expenses, you'll often come across a gross amount that hasn't had the tax separated out and you'll have to get your calculator and work out NET and VAT for yourself before typing them in. A macro can do all this work for you.

Let's start as usual, by doing the operation by hand and letting Cardbox record it. Start editing a record and put an amount (for example 54.99) into its TOTAL field. Then turn on the macro recorder, type sample amounts into the NET and VAT fields, and turn the macro recorder off again. You'll get something like this:

```
GoToField "NET"
TypeText "46.80"
```

```
    GoToField "VAT"
    TypeText "8.19"
```

As it stands, this macro would only work if the amount was always 54.99, so we need to rewrite it and make it perform a calculation. Here is the rewritten version:

“1.175” is because at the time of writing the rate of VAT in the UK is 17½%. For a full explanation of how this macro works, see Help Point 755,

```
    amount=Fields("TOTAL")+0
    net=Round(amount/1.175,2)
    GoToField "NET"
    TypeText FormatNumber(net,2,True,False,False)
    GoToField "VAT"
    TypeText FormatNumber(amount-net,2,True,False,False)
```

Here is what the macro does, step by step. All the features will be covered in detail later on in this book.

1. `Fields("TOTAL")` is a Cardbox function that gets the value in the current record’s TOTAL field. Adding 0 is a quick way of turning a piece of text into a number that can be used for calculations.
2. `Round()` is a VBScript function that rounds a value to a given number of decimal places (in this case, two).
3. `GoToField` is a Cardbox command that moves to a named field.
4. `FormatNumber()` is a VBScript function that converts a number to text. VBScript can do this automatically but `FormatNumber` gives more control: in this case, it ensures that the result always has two decimal places and that it has a leading zero where necessary (so that 0.8 becomes “0.80” and not “0.8” or even “.8”).
5. `TypeText` is a Cardbox command that types text into a field.

This all looks quite technical but you only need to set it up once. You can attach the macro to a single keystroke such as Ctrl + Alt + V. After that, you can enter the TOTAL field by hand when you are adding a record, and then just press a key to get the macro to fill in NET and VAT for you. The effort you put into writing the macro will pay for itself very quickly.

Example: processing a selection of records

Batch Edit effectively works as an “instant macro” and this would interfere with the standard macro mechanism, which is why it isn’t allowed when recording macros.

If you are using Cardbox directly and want to edit a whole batch of records in the same way, you select them and use the Edit > Batch > Edit command. This command isn’t available when you’re recording a macro but you can easily adapt a macro that edits every record in a selection. This goes beyond what Batch Edit can do, because the macro can do things that are a lot more sophisticated than simply repeating keystrokes.

Let’s go back to the tax example above. Suppose that instead of doing the tax calculation while entering a record, you want to enter all the records first, then select the ones that need the calculation and process them in a batch. What you do is take our original example and add some lines to the top and bottom:

```
For pos=1 to RecordCount
GoToRecord pos
EditRecord

    amount=Fields("TOTAL")+0
    net=Round(amount/1.175,2)
    GoToField "NET"
    TypeText FormatNumber(net,2,True,False,False)
    GoToField "VAT"
    TypeText FormatNumber(amount-net,2,True,False,False)

SaveRecord
Next
```

Here's what is going on.

This macro will work correctly even if there are no records in the current selection: RecordCount will be 0 and whole For / Next loop will do nothing.

1. `RecordCount` is a built-in Cardbox function that gives the number of records in the current selection.
2. The `For` / `Next` loop performs the commands inside it repeatedly, for `pos=1`, `pos=2` and so on, up to `pos=RecordCount`.
3. `GoToRecord` moves Cardbox to the given record.
4. `EditRecord` starts to edit the record.
5. Now the original macro takes over and does the same calculations it did before.
6. `SaveRecord` saves the record after editing.
7. `Next` repeats the process as many times as necessary.

Using this process for your own macros

Whenever you have a macro of your own that just processes one record, you can turn it into one that processes the whole of the current selection. Just add lines to the top and bottom of it in the way that has been shown in the example.

Part Three
Using VBScript

Useful tools; basic commands; variables; date and number conversions; objects and object lifetimes.

This chapter gives you an overview of VBScript but it isn't a step-by-step tutorial. If you are familiar with other dialects of Basic, the information here will be quite enough to get you going. If you aren't, we recommend reading it in conjunction with the sample macros in this book. The information here is very dense and you shouldn't expect to absorb it all on a first reading.

For some suggested books, see page 55 or Help Point 750.

You may also need to look further afield. Microsoft's web site has a complete reference to the language; there are a number of tutorials on the Web: use a search engine to find them. A number of books on VBScript are also available.

Data types

The data types that VBScript handles are numbers, strings, dates, and objects. Variables in VBScript don't have a predefined type: any variable can store any type of data.

Numbers

The basic numeric operators are the same as in other programming languages:

Addition	`+`	`2 + 2` is 4 (but see the warning on page 28)
Subtraction	`-`	`3.14 - 2.78` is 0.36
Multiplication	`*`	`37 * 3` is 111
Division	`/`	`100 / 8` is 12.5
Parentheses	`()`	Calculations inside parentheses are done before calculations outside parentheses, so: `1 + (2 * 3)` is 7 `(1 + 2) * 3` is 9
Equals	`=`	`2=3` is False
Does not equal	`<>`	`2<>3` is True
Greater than	`>`	`2>3` is False
Greater than or equal	`>=`	`2>=3` is False
Less than	`<`	`2<3` is True
Less than or equal	`<=`	`2<=3` is True

WARNING: If you use + to add two fields then you will get unexpected results. If the field AA contains “2” and the field BB contains “3”, then

```
Fields("AA") + Fields("BB")
```

The revised version works because if VBScript sees a number on one side of + and a string on the other, it treats + as addition. See Help Point 766.

will give you “23”. This is because the field values are strings, and if VBScript sees a string on each side of the + then it concatenates the two strings. To avoid this, insert a zero between the two field references:

```
Fields("AA") + 0 + Fields("BB")
```

There are also some functions that help to get the results of calculations into the format you want. Here are the most useful ones

For more details of how these functions work, and a complete list of mathematical functions, see Help Point 756.

`Int(`*value*`)`	The integer part of a number.	`Int(7/3,2)` is 2
`Fix(`*value*`)`	Similar to `Int`, but works differently for negative numbers.	
`Round(`*value*`,`*places*`)`	Rounds a number up or down to a given number of decimal places.	`Round(7/3,2)` is 2.33 `Round(8/3,2)` is 2.67
`CInt(`*value*`)`	Rounds a number to an integer.	`CInt(2.5)` is 2 `CInt(2.6)` is 3

Strings

A string is a sequence of characters: the minimum length of a string is 0 (this is the empty string, `""`) and the maximum length is millions of characters.

To specify a string, just put it in quotation marks.

```
MsgBox "Hello, world!"
```

will display

```
Hello, world!
```

To include a quotation mark within the string, type it twice:

```
MsgBox "Welcome to the ""machine"""
```

will display

```
Welcome to the "machine"
```

There is only one useful string operator:

+ is a synonym for & but we don’t recommend that you use it because it has a different meaning when used with numbers.

Concatenation	&	`"AB" & "CD"` is “ABCD”

There are functions for manipulating pieces of a string. In the examples shown, we assume that `x` represents the string “COOPERATE”:

`Left(`*string*`,`*count*`)`	The first characters of the string.	`Left(x,4)` is “COOP”
`Right(`*string*`,`*count*`)`	The last characters of the string.	`Right(x,3)` is “ATE”
`Mid(`*string*`,`*start*`,`*length*`)`	Characters in the middle of the string.	`Mid(x,3,5)` is “OPERA”
`Mid(`*string*`,`*start*`)`	Characters starting at a given point in the string.	`Mid(x,3)` is “OPERATE”
`Len(`*string*`)`	The length of the string.	`Len(x)` is 9
`Instr(`*string*`,`*pattern*`)`	The position of a pattern within the string.	`Instr(x,"ER")` is 5 `Instr(x,"ET")` is 0

`Instr(`*pos*`,`*string*`,`*pattern*`)`
`Instr(`*pos*`,`*string*`,`*pattern*`,`*mode*`)`
	Extra options in `Instr` let you specify where the search should start, and whether it should be case-sensitive or case-blind. See Help Point 757.	

There are some built-in constants and functions to help you put together strings that aren’t easy to type:

`vbLf`	The line feed character. Cardbox uses this to separate the lines of a multi-line field.	`vbLf` is the same as `Chr(10)`
`vbCrLf`	A newline sequence. This is used to separate lines in Windows files.	`vbCrLf` is the same as `Chr(13) & Chr(10)`
`Chr(`*code*`)`	The character whose Windows code number is *code*.	`Chr(68)` is “D” `Chr(224)` is “à”
`ChrW(`*code*`)`	The character whose Unicode code number is *code*.	`ChrW(321)` is “Ł”
`Asc(`*char*`)`, `AscW(`*char*`)`	The opposite of `Chr` and `ChrW`	

There are functions to remove leading and trailing spaces from a string. In the examples shown, we assume that `x` represents the string “ XXX ”.

`LTrim(`*string*`)`	Removes spaces at the beginning of the string.	`LTrim(x)` is “XXX ”
`RTrim(`*string*`)`	Removes spaces at the end of the string.	`RTrim(x)` is “ XXX”
`Trim(`*string*`)`	Removes spaces at the beginning and end of the string.	`Trim(x)` is “XXX”

There are functions to convert a string to upper or lower case:

`LCase(`*string*`)`	Converts the string to lower case.	`LCase("Ab")` is “ab”
`UCase(`*string*`)`	Converts the string to upper case.	`UCase("Ab")` is “AB”

String comparison

The standard comparison operators <, =, >, <=, <>, >= have the same meanings that they do for numbers.

Comparison happens one character at a time. If one string ends before the other, and they are identical up to that point, the longer string is counted as being greater than the shorter. For example, `"Rat"<"Rate"` is `True`.

Comparison using these operators is case-sensitive. This means that `"Rat"="RAT"` is `False`. In fact, `"Rat">"RAT"`, because lower-case letters are usually greater than upper-case letters.

If your macro will be used on Turkish text, see Help Point 758.

To do case-blind comparison, you have two choices. You can use `LCase`:

`LCase("Rat") = LCase("RAT")` is True

StrComp is said to be more efficient, but in general you should prefer readability to efficiency when writing macros.

or you can use `StrComp`, which returns 0 for equality, -1 if the first string is less than the second, and 1 if the first string is greater than the second:

`StrComp("Rat","RAT",vbTextCompare)` is 0

`StrComp("Rat","SAT",vbTextCompare)` is -1

`StrComp("Rate","RAT",vbTextCompare)` is 1.

Dates

VBScript can handle dates from 1 January 100 to 31 December 9999.

The function `Date` always returns the current date. Various other functions can be used to create specific date values: see "Dates and Strings" on page 35.

The following functions are relevant to dates. We'll assume that x has a date value of 16 October 1978.

d+*n*	*n* days after the date *d*.	`x+200` is 4 May 1979
d-*n*	*n* days before the date *d*.	`x-51` is 26 August 1978
date2-*date1*	The number of days from *date1* to *date2*.	`x-DateSerial(3,8,1923)` is 20163
`DateSerial(`*d*`,`*m*`,`*y*`)`	The date with day *d*, month *m*, and year *y*.	
`Day(`*date*`)`	The day of the month of *date*.	`Date(x)` is 16
`Month(`*date*`)`	The month of *date*.	`Month(x)` is 10
`Year(`*date*`)`	The year of *date*.	`Year(x)` is 1978
`Weekday(`*date*`)`	The weekday of date, with Sunday=1. See Help Point 759 for more options.	`Weekday(x)` is 2
`DateAdd("`*code*`",`*n*`,`*date*`)`	n units of time after *date*. **"code"** **Units** `"yyyy"` Years `"m"` Months `"d"` Days	`DateAdd("m",3,x)` is 16 January 1979
`DateDiff`	The difference between two dates. See Help Point 759.	

Arrays

Arrays in VBScript work as they do in most other dialects of Basic. The statement

```
    Dim arr(3)
```

If the array index is outside the allowable range, VBScript will report an error.

creates an array of four values, called `arr(0)`, `arr(1)`, `arr(2)` and `arr(3)`. The useful thing about arrays is that the array index can be a variable or a numeric expression: so `arr(ix)` will refer to one of the four array elements depending on the value of `ix`.

Two VBScript functions, `LBound` and `UBound`, give the minimum and maximum allowable array index. In the example we've given, `LBound(arr)` will be 0 and `RBound(arr)` will be 3. In that case the functions are telling you something you already know because you typed the `Dim` statement yourself; but this isn't always the case. For example, the `GetMailExchangers` method provided by Cardbox creates and returns an array value and you'll want to use `LBound` and `UBound` to find out how many elements the array contains.

Splitting and joining strings

The built-in VBScript function `Split` splits a string into an array of shorter pieces. You give it the string to be split, and the characters that are being used to separate the pieces. For example:

```
    slices=Split("Alpha,Bravo,Charlie",",")
```

creates an array with the following attributes:

```
    slices(0)="Alpha"
    slices(1)="Bravo"
    slices(2)="Charlie"
    LBound(slices)=0
    UBound(slices)=2
```

The useful thing about this is that if you receive some data in the form of a string, you can split the string into an array and then loop through the array elements one at a time, processing each one in whatever way you want. Here are some examples:

- A multi-line Cardbox field is a string with line feed characters (`vbLf`) used to separate each line from the next; so `Split(Field("ADDR"),vbLf)` will be an array in which each line of the field called ADDR is a separate element. You may, for example, use this fact if you are using a macro to output data in some specialised format.
- Cardbox's `ListIndex` method returns a list of matching index terms as a single string separated by newline (`vbCrLf`) markers. So if you want to process each term individually, store the result of `ListIndex` in a string (let's call it `ixlist`)

and then use `Split(ixlist,vbCrLf)` to split it into an array of separate index terms.

- Windows text files consist of lines separated by newline markers. One technique for processing a file is to read it all into a single string and then use `Split` to split it into lines.

The opposite of `Split` is `Join`. `Join` joins an array together into a single string. To take one example: given the `slices` array that we created using `Split`, using `Join(slices," ")` will give you the string `"Alpha Bravo Charlie"`. Combining `Join` with `Split` gives you a quick way of changing lists from one format to another.

Conversions

Numbers and strings

If you use a string in a numerical context and it doesn't represent a valid number then VBScript will report an error.

The string `"123"` is a string and the number `123` is a number: they look similar to us but to the computer they are two different things. VBScript covers up the differences quite efficiently: if you use a string where a number is expected (for example, in an arithmetical calculation) it will convert the string to a number; if you use a number in a context where a string is required, it will convert the number to a string.

For example:

```
"27" + 0
```

has the numerical value 27: this is a quick and easy way of forcing a numerical value when you need one. And

```
MsgBox "Total = " & 199.90
```

will display the message

```
Total = 199.9
```

There is an additional function that gives you more control over the conversion of numbers to strings: this is `FormatNumber`.

`FormatNumber(`*value*`,`*places*`,`*lzero,paren,group*`)` formats *value* to have exactly *places* decimal places.

- If *lzero* is `True`, then if *value* is less than 1, `FormatNumber` puts a zero before the decimal point.
- If *value* is negative, `FormatNumber` puts a minus sign in front of it if *paren* is `False` or parentheses round it if *paren* is `True`.
- If *group* is `True`, `FormatNumber` groups the digits (eg. 2,000 rather than 2000).

Here are a few examples to make it clearer:

- `FormatNumber(123,2,True,False,False)` is “123.00”.
- `FormatNumber(123.456,2,True,False,False)` is “123.46”.
- `FormatNumber(-0.5,3,False,False,False)` is “-.500”.
- `FormatNumber(0.1,3,True,False,False)` is “0.100”.

Regional settings

There is one qualification to all of the above description: the description is true only if the Regional Settings of your computer (set in the Windows Control Panel) indicate that the decimal separator in your part of the world is a decimal point. If you are somewhere where the decimal separator is a comma (and Windows knows this) all the conversions shown above will use commas instead of decimal points. If this is what you want, there is nothing to worry about. But Cardbox has to be consistent across the world and so, when it indexes, always assumes that the decimal separator is a decimal point. Thus if you use VBScript functions to convert numbers into strings and then pass them to Cardbox, you may run into trouble: run in Belgium, for example, your macro will convert `3.14` to `"3,14"`, which Cardbox won’t understand.

In selection commands, Cardbox will reject “3,14” as a search string. When indexing data, it will interpret “3,14” as the number 314.

There are two cases where you won’t have difficulty.

- If you pass a number into a Cardbox search command without converting it into a string first, Cardbox will do the conversion for you and always use a decimal point so that everything works.
- If you assign a number directly to a Cardbox field value – for example, `Fields("TAX")=144.62` – or pass a number directly to the `TypeText` method, Cardbox will also perform the conversion for itself.

To handle other cases (for example, where VBScript has to build a string and then pass it to Cardbox) there are two functions built into Cardbox’s macro system:

`NumberToCardbox(`*value*`)`	Converts the number to a string, always using a decimal point as the separator.	`NumberToCardbox(3.14)` is “3.14”
`NumberFromCardbox(`*value*`)`	Converts the string to a number, always using a decimal point as the separator.	`NumberFromCardbox("3.14")` is 3.14

Dates and strings

VBScript provides built-in functions to convert between dates and strings:

`DateSerial("`*string*`")`	Converts a string to a date.	
`FormatDateTime(`*date*`)`	Converts a date to a string.	

Regional settings

Unfortunately there are even more regional settings for dates than there are for numbers, and many of the settings don't fit Cardbox's pattern of `day/month/year`, `month-day-year`, or `year.month.day`

- If you pass a date into a Cardbox search command without converting it into a string first, Cardbox will do the conversion for you.
- If you assign a date directly to a Cardbox field value or pass a number directly to the `TypeText` method, Cardbox will also perform the conversion for itself, using the `day/month/year` format.
- If you assign a number directly to a Cardbox field value – for example, `Fields("TAX")=144.62` – then Cardbox will also perform the conversion for itself.

For all other cases, there are two conversion functions:

`DateToCardbox(`*date*`)`	Converts the date to a string, always using a slash as the date separator.
`DateToCardbox(`*date*`,"`*sep*`")`	Converts the date to a string, using the separator "*sep*" (`"/"`, `"-"`, or `"."`).
`DateToCardbox(`*date*`,"`*pattern*`")`	Converts the date to a string in virtually any format: see Help Point 760.
`DateFromCardbox("`*string*`")`	Converts the string to a date. The string may be in any of Cardbox's standard date formats.

Two-digit years

Cardbox is quite comfortable with two-digit years as long as you don't mix them with four-digit ones: to Cardbox, `68` means the year 68 and `1968` means the year 1968.

VBScript can't handle years before 100AD, and it translates a two-digit year into a year in the 20th or 21st century. Different versions of the Windows system files make different assumptions about when a two-digit year should be put into the 20th and when into the 21st century, and it is said that Microsoft's documentation has at times been inconsistent and not matched the reality. We strongly recommend that if you are going to use two-digit year numbers in your system then you should add 1900 or 2000 to them yourself before passing them to any of VBScript's date functions. That way you will know exactly what to expect.

Objects

VBScript interacts with other programs by using objects. Objects are abstract entities that can represent anything outside VBScript. To take some examples:

A File object (created by the FileSystemObject component of Windows) lets you manipulate a file; a Folder object lets you manipulate a folder.

A Window object (created by Cardbox) lets you send commands to a Cardbox window, and change things like its caption and position; a Field object gives you access to the contents of a single field.

A WinFax object (created by Symantec's WinFax Pro) lets you send commands to the fax system and check its status.

Splitting the world into objects can seem inconvenient at first but it has the great advantage that VBScript doesn't have to understand anything about the programs it is communicating with: to VBScript, everything is an object.

Creating objects

There are two basic ways of creating objects:

- In VBScript, the `CreateObject` and `GetObject` functions can create an object.
- Many objects have commands ("methods") that allow the creation of sub-objects. For instance, to get a Folder object for manipulating a Windows folder, you use `CreateObject` to create an object of type FileSystemObject and then use that object's `CreateFolder` or `GetFolder` method to get hold of the Folder object.

Whenever we document an object type in this book we'll also tell you how to create an object of that type.

Methods and properties

Methods

A method is a command provided by an object. Often the command will change the object in some way – for instance, the `Select` method of Cardbox's Window object performs a selection command in a window – but that isn't essential: some methods are just a way in which an object provides a service to your macro. Strictly speaking, the conversion functions that Cardbox provides (such as `DateFromCardbox`, p. 35) work like this: they are actually methods of an invisible built-in Macro object: the `DateFromCardbox` method receives a string value from your macro and returns a Date value to it.

Properties

A property is a fact about an object. For instance, the `Text` property of the Field object is the actual text of the field; the `Caption` property of the Window object is the window's caption. Properties can have any type of value (number, string, etc) and they can be read/write (modifiable by a macro or program) or read-only. To give one example: the `Text` property of the Field object is read/write if the record is being edited and read-only if it isn't.

Objects from objects

There is a diagram of the Cardbox object model on page 58. If you know the name of a method or property and want to know which objects it applies to, see the Index of Methods and Properties on page 132.

Object methods can return an object as their value: for example, the FileSystemObject object provided by Windows has a method called `GetFile` which creates an object that refers to an existing file. This is the normal way of navigating through a program's "object model" and Cardbox uses it a lot. Given a Window object, you can get a Records object that represents the objects in that window; a Record object that represents one of those records; and a Field object that represents one field in that record. You'll see how this works later in this book, when we show you some examples.

Collections

Some objects represent collections of other objects. To take an example within Cardbox: a Records object represents a collection of Record objects – depending on where you got the Records object from, these might be all the records in the database, or all tagged records, or all the records in the current selection.

All collection objects have the following standard properties:

`Count`	The number of objects in this collection.
`Item(`*n*`)`	The *n*th object in this collection.

Example

Suppose that you have a Records object called `recs`. Then `recs.Count` will be a number telling you how many records there are in `recs`, and `recs.Item(3)` will be a Record object representing the third record in `recs`. It would be an error to refer to `recs.Item(3)` if `recs` contained fewer than three records.

- In most programming languages (including VBScript) you can say `recs(3)` instead of `recs.Item(3)` and it will mean exactly the same thing.
- In some collections, the index *n* can be a string as well as a number. For example, if `flds` is a Fields object, `flds.Item("NAME")` or `flds("NAME")` will be a Field object representing the field whose name is NAME.

Variables

A variable is something that holds a value. If you say

```
x = "world"
MsgBox "Hello again, " & x & "!"
```

then the macro will display

```
Hello again, world!
```

You can replace the contents of a variable as often as you like:

```
number = 2
number = number + 3
MsgBox number
```

first sets `number` to 2, then calculates 2+3 and stores the result back into `number`, then displays 5 as the result. This example is pretty pointless, but repeatedly adding numbers to a variable is exactly what you need to do if you are calculating totals.

Variables in VBScript don't have a fixed type, so you can use a variable to store a number at one moment and a string a moment later; although it's usually best for the readability of your macros if you don't make a habit of that sort of thing, because it can make it harder for the reader (who might be you in a few months' time) to make out what is going on.

Variable names

A variable name must start with a letter and contain only letters, numeric digits, and the underline character "_". You can have variable names more or less as long as you like: the limit is over 200 letters.

VBScript ignores case in variable names, so `recSelected` and `RecSelected` and `recselected` and `RECSELECTED` are all the same name as far as VBScript is concerned, but it's a good idea to establish some sort of convention so that you don't confuse yourself. We tend to do the following:

`RecSelected` **capitalisation** is used for methods, properties, functions and subroutines.

`RECSELECTED` **capitalisation** is used for values that are constant throughout the macro: for example, `const TAXRATE=0.175`. This makes it easy to see where the tax rate is being used and easy to change it if necessary.

`recSelected` **capitalisation** is used for all object references. We use prefixes to remind ourselves what kind of object the variable ought to contain: things like `win`, `recs`, `rec`, `flds`, `fld`, and so on. We also sometimes use the prefix `i` for numbers that indicate position in a list and the prefix `n` for numbers that indicate a count of some kind. It's not compulsory but it helps readability.

A simple uncapitalised name is used when there's no possibility of confusion. If there's only one variable in the macro that holds a Record object, and it's clear what Record object is involved, then there's no harm in just calling the variable `rec` on its own

For variables that have numeric or string values, we sometimes use short uncapitalised names (`amount`, `tax`), multi-word names with all words capitalised except the first, or two-word names with no capitals at all. One-letter names are reserved for variables that are only used in a restricted context: within a few adjacent lines of the macro, or within a single loop.

Remember: the primary purpose of any program (and macros are small programs) is to convey its intention and structure to a future reader. The fact that a program does something when run is simply a happy side-effect. Use our rules or work out your own: it doesn't matter which, but do think of the person who may have to modify, imitate or debug your macro in the future.

Avoiding keywords

A variable name must not be the same as a VBScript keyword, or VBScript will get confused. So you can't have a variable called `next` or `date`, because `Next` is a VBScript command and `Date` is a VBScript function. `nextRecord` or `dateOfBirth` are perfectly OK. Similarly, a variable name mustn't be the same as one of the built-in methods or properties that Cardbox adds to the macro environment: these are listed on pages 111 and 132.

Storing object references in a variable

To set the value of a variable, you use =:

```
nHappyCustomers = nHappyCustomers + 1
name = "Cletus"
```

but objects are an irritating exception. To set a variable to contain an object reference, you have to use `Set`:

```
Set fso = CreateObject("Scripting.FileSystemObject")
```

If you don't use `Set`, you'll end up storing a rather nebulous entity called the "default value" of the object instead of the object reference you were expecting. For some objects, this is rational: the default value of a Field object is the text of the field; but most objects don't have meaningful default values and you'll get an error if you forget to use `Set`.

`CreateObject` creates an empty object of a specified type. To create an object that refers (for example) to a Word document file, use `GetObject` instead. For example:

```
Set doc = GetObject("C:\Docs\MyLetter.doc")
```

Scrap variables

Normal variables are part of a macro and as soon as the macro terminates, all its variables disappear. Some people like to create a set of interrelated macros that can share data values between them. If you are one of these people then the Scrap variables are what you need. The variables `Scrap(1)`, `Scrap(2)`,… up to `Scrap(100)` are maintained by Cardbox rather than by the macro, so any values that you give to them will persist for as long as this copy of Cardbox is running.

Flow of control

Unless otherwise specified, VBScript executes a macro by beginning at the beginning and going on until it reaches the end. All automatically-recorded macros are of this form. But when you're adding intelligence you will probably want to add loops (to process a batch of records) and conditional logic (to do different things in different cases). Here are the most commonly used constructs for altering the flow of control.

Halt

This isn't a VBScript statement but a Cardbox method call; but it's important enough to be included here.

`Halt` on its own terminates the macro.

`Halt "`*message text*`"` terminates the macro and displays a message. Use it, for example, if there is something wrong and you want to let the user know.

For / Next

If you are used to other dialects of Basic, you will expect to say "`Next i`" or "`Next iRec`" rather than just "`Next`". Unfortunately VBScript only allows a plain "`Next`" and it reports an error if you try to include a variable name as well.

```
For i  = 1 To 10
 VBScript statements
 Next
```

executes the VBScript statements between For and Next once for `i=1`, once for `i=2`, and so on up to `i=10`.

The starting and ending points don't have to be fixed numbers: they can be any numeric expression:

```
nRecs=RecordCount
For iRec=1 To nRecs
 VBScript statements
 Next
```

In this case the loop variable is called `iRec`, and the loop will be executed once for `iRec=1`, once for `iRec=2`, and so on up to `iRec=nRecs`. If `nRecs=0` then this isn't an error: VBScript will just skip over the loop.

The use of `nRecs` is not a piece of gratuitous bureaucracy. VBScript checks "have I come to the end?" every time it goes through the loop, and since `RecordCount` is a built-in Cardbox property, this means communicating repeatedly with Cardbox, which can slow things down (especially if there are thousands or tens of thousands of records). In most cases we don't recommend making a macro more complicated for the sake of making it faster, but this is one of the exceptions.

This rule applies to all block statements: For/Next, Do While/Loop, If/End If, Sub/End Sub, and so on. They can all contain other blocks and all (except for Sub and Function) can be nested inside other blocks; but you must always close the most recently opened block first.

- You can make a loop that steps through its values in bigger jumps: for example, `For i=1 To 6 Step 2` would execute the loop statements for `i=1`, `i=3`, and `i=5`. This isn't often useful.
- You can make a loop that goes backwards: For i=10 To 0 Step -1 will count down from 10 to 0, inclusive. This is useful sometimes.
- The loop can itself contain another loop, and that loop can contain yet another one. The only rule is that you have to close the loops in the opposite order to their opening – so that the most recently opened loop gets closed first:

```
For i  = 1 To 10
 VBScript statements
 For j = 1 To 6
  VBScript statements
  Next
 VBScript statements
 Next
```

- If your macro discovers, in the middle of a loop, that it wants to leave the loop, it can use the statement `Exit For`. This immediately jumps to just after the `Next` statement of the loop it is in.

For Each / Next

This is almost exactly the same as `For / Next`, but it loops through a collection of objects. It is best illustrated by an example:

```
Set recs = ActiveWindow.Records
total = 0
For Each rec In recs
 total = total + rec.Fields("AMT")
 Next
```

This sets `recs` to be a Cardbox Records object, which is a collection (p.37) of Record objects. The effect of `For Each` is to go through the loop once for each object in the collection, with `rec` having a reference to each of the Record objects in turn. The overall effect of the example is to calculate the total of the AMT field in every record in the current selection and store it in the variable called `total`.

Do While / Loop

This is a simple kind of loop that doesn't set any variables, just executes repeatedly until a condition is true:

```
Do While winfax.IsEntryIDReady(0) <> 1
  Sleep 20
  Loop
```

We didn't invent this code ourselves but took it from a VBScript example provided by Symantec. When dealing with external objects, the manufacturer's own documentation is always the best guide.

This is an extract from a macro for sending faxes using Symantec's WinFax Pro. `winfax` is an object that represents the WinFax program. The macro has just sent a command to send a fax, and now it has to wait until the object reports that the program has finished processing the command (if it doesn't do this, WinFax will get extremely confused). So the macro repeatedly checks the value returned by WinFax's `IsEntryIDReady` method, and pauses for 1/50 of a second between each check so as to give WinFax a chance to do some actual work.

- The loop will repeat as long as the `Do While` condition is satisfied. It will repeat forever if necessary.
- If the `Do While` condition isn't satisfied the first time the statement is reached, the loop is not executed at all and VBScript goes straight to the statement after `Loop`.
- If your macro discovers, in the middle of a loop, that it wants to leave the loop, it can use the statement `Exit Do`. This immediately jumps to just after the `Loop` statement of the loop it is in.
- The condition after `While` can be a comparison (see page 37) or a simple numeric expression, in which case a non-zero value is counted as `True` and zero as `False`.
- Complex conditions can also be built up using the logical operators `And`, `Or`, and `Not`. The way that VBScript interprets these operators is not very intuitive and we recommend that you put parentheses round the conditions that you are joining together. For example, `(x=3) Or (x=5)` is true if x equals 3 or 5.

While / Wend

This is identical to `Do While / Loop` but is out of fashion.

Do Until / Loop

This is the opposite of `Do While`. Instead of looping while a condition is satisfied, it loops while the condition isn't satisfied – which is the same thing as looping until the condition is satisfied, which is how this loop gets its name.

Do / Loop While

This is similar to `Do While / Loop` but there is a subtle difference:

```
Do
  VBScript statements
  Loop While MsgBox("Try again?",vbYesNo)=vbYes
```

In this case the `While` test takes place at the end of the loop rather than at the beginning, which means that the statements inside the loop will always be executed at least once. The example shows a loop which will execute some statements, then ask the user whether he wants to try again, and will repeat the statements as long as he answers “Yes” each time.

- One example of where this sort of thing is useful is if the body of the loop asks for a password and then tries to check if it works: if the password does work, the macro can use `Exit Do` to leave the loop and stop asking the user whether to try again.

Do / Loop Until

This is the opposite of `Do / Loop While` and it loops as long as the `Until` condition is false (or, if you prefer, stops looping as soon as the `Until` condition becomes true).

If / Then / Else / End If

The general form of a multi-line `If` statement block is:

```
If condition Then
  VBScript statements if condition is true
 Else
  VBScript statements if condition is false
 End If
```

If you’re not interested in what happens if *condition* is false, you can leave out the `Else` part altogether:

```
If Fields("TYPE")<>"T" Then
  MsgBox "Invalid type"
  Halt
  End If
```

There is nothing to stop you having `If` statements inside `If` statements: in fact, any kind of block (`For`, `Do`, `While`, `Select`, `If`) can be nested inside any other.

If / Then / Else

You can also write an `If` statement on a single line:

```
If condition Then VBScript statements Else VBScript statements
```

In this case End If isn't needed:

```
If Fields("TYPE")<>"T" Then MsgBox "Invalid type" : Halt
```

Note that you can use a colon to put several statements together on a line to take advantage of this abbreviated form of `If`. Don't overdo this: as always, legibility is the most important thing.

Select Case / Case / End Case

The final conditional construct in VBScript is the most elaborate:

```
Select Case value
 Case testvalue
  VBScript statements
 Case testvalue,testvalue
  VBScript statements
 Case Else
  VBScript statements
 End Select
```

(Each `Case` can have one or more test values, but if there is more than one value then they need to be separated by commas).

Here is what VBScript does with `Select Case`:

1. It calculates *value* and then looks at each `Case` in turn.
2. It calculates each of the test values given and compares it to *value*. If they match, it executes the VBScript statements that follow, up to but not including the next `Case`, and skips to just past `End Select`.
3. `Case Else` is optional. If it is present, it must be the last case: it will match any *value* that didn't match any of the other `Case` lines.

Sub / End Sub

Sub / End Sub defines a subroutine: that is, a set of VBScript statements that you can then call using a single statement. There are thus two parts to the use of a subroutine: you need to define it, and you need to call it.

You can put the subroutine definition before the code that actually calls the subroutine, or after it: choose whichever style you find more readable.

Subroutine definition

```
Sub SubroutineName(param1,param2,…)
 VBScript statements
 End Sub
```

- You choose the name of the subroutine yourself. The rules are the same as for variable names (p.38).
- Parameters enable the caller of the subroutine to pass values to it. There can be any number of parameters (you choose their names too): if there is more than one then you should separate them with commas. Subroutines without parameters are also possible: in that case you don't need the parentheses at all.
- You can use `Exit Sub` within a subroutine to cause an immediate exit from the subroutine back to the statement that called it.

Subroutine calls

There are two forms:

```
SubroutineName arg1,arg2,…
```

and

```
Call SubroutineName(arg1,arg2,…)
```

They do exactly the same thing, so choose whichever looks better to you. Visually, the first form looks more like a VBScript statement and the second form looks more like a function call.

- When you call a subroutine, you supply one value ("argument") for each parameter that was listed in the subroutine definition. The parameters will be given the values you supply, and the statements in the subroutine will then be executed until the end of the subroutine is reached or an `Exit Sub` statement is executed. After that, the macro will continue with the statement after your subroutine call.
- Subroutines can call other subroutines.

Examples

```
Sub ReportErrorAndHalt(msg)
 MsgBox msg & vbCrLf & "For help, please call extension 43."
 Halt
 End Sub
```

This subroutine is designed to report an error message, which is given to it as an argument. It displays the message together with a contact telephone number that the user can call for assistance, and then halts the macro. Here are two sample uses of the subroutine:

```
ReportErrorAndHalt "No records were selected."
ReportErrorAndHalt "This quarter's report is not yet ready."
```

Subroutines can also be used to improve legibility. For example, in the following macro (which loops through an entire selection using `For` / `Next`), the subroutine call clearly separates the business of looping through the records from the actual processing that is done to each record:

```
Sub ProcessRecord
 EditRecord
 VBScript statements
 SaveRecord
 End Sub

For i=1 To RecordCount
 GoToRecord i
 ProcessRecord
 Next
```

Function / End Function

Functions are like subroutines except that they return a value that you can store in a variable or use in calculations. To return a value from a function, you store the value into the function name. Here is a very simple example:

```
Function PlusOne(x)
 PlusOne = x + 1
 End Function

MsgBox PlusOne(18)
```

will display “19”.

Here is a less trivial function:

```
Function ZeroIfBlank(str)
 x = Trim(str)
 If x = "" Then ZeroIfBlank = 0 : Exit Function
 ZeroIfBlank = x + 0
 End Function

MsgBox ZeroIfBlank(Fields("VALUE"))
```

This function is very useful if you are doing numerical work with Cardbox. It converts a value (typically, the contents of a field) to a number, except that if the value is blank or contains nothing but spaces, it converts it to zero. (If a string is blank, VBScript’s own built-in number conversions would report an error if you tried to convert it to a number).

Variable names in subroutines and functions

Subroutines are to some extent isolated entities as far as variable names are concerned, but the exact rules are a little abstruse:

1. If you declare a variable inside a subroutine by using the `Dim` statement, then that variable is private and is known only inside the subroutine. You can use the same name elsewhere in the macro if you feel like it, but it won't refer to that variable.
2. If you don't declare the variable inside the subroutine, but it doesn't get used in the body of the macro, then it is also private. By "the body of the macro" we mean the part of the macro text that isn't inside any subroutine or function.
3. If you don't declare the variable inside the subroutine, and it does get used in the body of the macro, then the variable is shared between the subroutine and the macro, so that if one of them makes a change to the value then the other will see the changed value.

Here is a rather artificial example:

```
Sub Pointer
 Dim x
 x=1 : y=2 : z=3
 End Sub
Sub Pug
 MsgBox x : MsgBox y : MsgBox z
 End Sub
x=8 : y=9
Pointer
Pug
```

- The value of `x` displayed by `Pug` will be 8, because the main body of the macro sets `x` to 8: the `x` used by `Pointer` doesn't count because it is declared inside Pointer and so changes made to it by `Pointer` are invisible outside `Pointer`.
- The value of `y` displayed by `Pug` will be 2, because `y` is used in the main body of the macro but not declared in either `Pug` or `Pointer`, so that `Pug` and `Pointer` and the body of the macro all refer to the same `y`.
- The value of `z` displayed by `Pug` will be blank, because `z` isn't used in the main body of the macro and so the `z` in `Pug` and the `z` in `Pointer` are both private and have nothing to do with each other: `z=3` inside `Pointer` affects `Pointer`'s `z` but not `Pug`'s.

It is difficult to remember all this and stay sane. The following pair of rules is all you really need:

1. If you want a variable to be private to a subroutine (which you usually do), use `Dim` to declare it inside the macro.

2. If you want a variable to be shared among all subroutines, use `Dim` to declare it in the body of the macro (a good place would be right at the beginning of the macro, before any subroutines).

User interaction

VBScript contains two functions for displaying messages and getting information from the user. These are useful but rudimentary: third-party components exist to allow VBScript to display more elaborate dialog boxes and a web search should find some suitable candidates if you need them.

MsgBox

You have already seen simple uses of `MsgBox` because we have used it as a way of displaying messages or reporting the result of a calculation. A fuller definition of `MsgBox` is:

`MsgBox` *message* , *options* , *title*

or

`x=MsgBox(`*message* , *options* , *title*`)`

Message specifies what will appear in the text of the message. It can be a string or anything that can be converted to a string. It can include the special `vbCrLf` code to split the message into separate lines or paragraphs.

Title is optional. It specifies what will appear in the title bar of the message box. It is also a string.

Options is optional. It is a combination of values that tell VBScript more about the message box. You can find a full list in Help Point 761, but the most useful values are those that specify which buttons should be shown in the message box:

`vbOKOnly`	OK
`vbOKCancel`	OK, Cancel
`vbYesNo`	Yes, No
`vbYesNoCancel`	Yes, No, Cancel
`vbAbortRetryIgnore`	Abort, Retry, Ignore
`vbRetryCancel`	Retry, Cancel

For an example, see the Do / Loop Until illustration on page 44.

If you use the subroutine form of `MsgBox`, you won't know what button the user pressed; if you use the function form, the value returned is `vbOK`, `vbCancel`, `vbYes`, `vbNo`, `vbAbort`, `vbRetry` or `vbIgnore`.

InputBox

The InputBox function displays a message asking the user for input, and returns the value that was entered.

`x=InputBox(`*message*`,`*title*`,`*default*`)`

Default is optional. If it is specified, the input box will appear with this value already fill in, so that the user can enter it just by clicking OK.

The input box has an OK button and a Cancel button. The return value of `InputBox` is the string the user typed if OK was pressed, or a zero-length string (`""`) if Cancel was pressed. This means that there is no way to distinguish between pressing Cancel and pressing OK with an empty box: both will result in `""`.

It is often useful to have a function that wraps InputBox with a few tests:

```
Function InputBoxOrHalt(prompt,dft)
 x=InputBox(prompt,"Cardbox Macro",dft)
 If x="" Then Halt
 InputBoxOrHalt=Trim(x)
 End Function
```

You call this function giving it a prompt and a default value. It displays a message using `InputBox` and checks the result to see if it's a zero-length string: if it is, it terminates the entire macro – this gives the user an opportunity to terminate the macro whenever he is asked for input. As a further convenience, this function also uses VBScript's `Trim` function to remove any leading or trailing spaces from what the user has typed.

Style and readability

See page 38.

We have already lectured you about making your macros readable, but we make no apologies for doing so again, because it's important. Here is a collection of suggestions for making macros readable and easily understood.

Indentation

When you are dealing with block-structured statements (such as For / Next or Sub / End Sub), use indentation to make it obvious to the reader where a block begins and ends. Our rules are:

In For loops, indent the statements inside the For by one space relative to the For, and indent the Next that closes the For in the same way:

```
For Each rec In recs
 total = total + rec.Fields("AMT")
 Next
```

- Apply the same rule to Do, While, Sub, and Function.

In `If` blocks, indent the `Else` (if present) and the `End If` by one space relative to the `If`, and indent the VBScript statements inside the block by two spaces relative to the `If` (by one space relative to the `Else` and `End If`).

- Apply the same rule to `Select` blocks.

Break any of the above rules if it makes the macro more legible.

Comments

The traditional Basic comment statement, REM, is also present in VBScript and you can use it as well if you like.

In VBScript, anything after an apostrophe is ignored:

```
Const VATRATE = 0.175
```

and

```
Const VATRATE = 0.175 ‘ UK rate current in 2004
```

are identical; but the second form may be more meaningful to a human reader.

You can also have comments on lines by themselves:

```
' Macro written by QHF on 20-Oct-2011.
' Note:
' The code dealing with multi-year renewals has been written
'  but not yet tested. Test it before use.
```

Blank lines

Blank lines cost nothing. They do nothing except make your macro easier to read. Use them.

Long statements

If you have a statement that is too long to read conveniently without scrolling, you can split it across more than one line. Simply end every line apart from the last with the underline character "_":

Make sure you don't put a space after the underline or VBScript won't see it as a continuation marker.

```
Records.WriteToFile cbxWriteFormatEXT, _
 "C:\Universalis\Programming\NEWS.EXT", _
 "ID,HEADING,NAME,**"
```

You can't split a character string across lines, but here is an alternative technique:

This separates the long string into two shorter ones joined by "&", and then splits the resulting statement across several lines.

```
Records.WriteToFile cbxWriteFormatEXT, _
 "C:\Universalis\Programming\source.ext", _
 "ID,NAME,RANK,DATE,USE,COMMON,ABBREVIATION," _
 & " FV_PS1,FV_PS2,FV_PS3,MAT_PS1,MAT_PS2,MAT_PS3”
```

Short statements

If you have several short statements then you can put them all on the same line by using colons:

```
total=0 : totalSelected=0 : previousCode=-1
```

Apart from its use in `If` statements, this is a dubious technique and we mostly avoid it because of the risk of illegibility. You can use it, though, if the separate statements all form part of a single action as far as a human reader is concerned.

Constants

The statement

```
Const RETRIES = 3 ‘ Number of retries for passwords.
```

gives RETRIES a value that can never be changed by the macro. It is useful because "RETRIES" is more noticeable than "3" when you are reading a macro, and it also clearly says what this particular use of the number 3 is for, which will save trouble and mistakes if the number of retries ever has to be altered.

We don't use constants in this way very much ourselves, but you will notice in many of our sample macros statements like

```
Const EMAIL_FIELD = "EMAIL" ‘ Name of the email field
```

This statement makes it much easier for you to adapt our sample to a database where the email address field has a different name.

Built-in constants

If you are using a different language from VBScript to drive Cardbox, you can still use Cardbox's predefined constants: see Help Point 762.

You will have noticed us using values such as `cbxWriteFormatEXT` or `vbYesNo` in our examples. These constants are built in to VBScript (the `cbx` constants are built in to the macro environment by Cardbox) and they are good because

```
    MsgBox("Try again?",vbYesNo)=vbYes
```

Is easier to read and easier to write than

```
    MsgBox("Try again?",4)=6
```

An important statement you should always use

An important statement that you should always use is:

```
    Option Explicit
```

If you put it at the very start of your macro then you will protect yourself from obscure errors caused by all sorts of typing mistakes.

Without `Option Explicit`, VBScript is informal: you don't need to do anything special when you want to use a variable, you just use it. With `Option Explicit`, VBScript is pedantic and you need to declare every single variable (with the `Dim` statement) before you use it. For example, here are the first few lines of a macro:

```
    For pos=1 to RecordCount
    GoToRecord pos
    EditRecord
    amount=Fields("TOTAL")+0
    net=Round(amount/1.175,2)
```

Without `Option Explicit`, this macro will work. With `Option Explicit`, VBScript will complain that you have used variables without declaring them, and you will have to put some extra lines at the beginning of the macro:

```
    Option Explicit
    Dim pos
    Dim amount,net

    For pos=1 to RecordCount
    GoToRecord pos
    EditRecord
    amount=Fields("TOTAL")+0
    net=Round(amount/1.175,2)
```

So why do we recommend all this extra bureaucracy? Look at this macro:

```
For pos=1 to RecordCount
GoToRecord pos
EditRecord
amount=Fields(TOTAL)+0
net=Round(amonut/1.175,2)
```

It looks the same as the one we showed you before, but it isn't. We've forgotten to put quotation marks round `TOTAL`, and we have typed `amonut` instead of `amount`. Without `Option Explicit`, VBScript will assume that we are simply using new variables called `TOTAL` and `amonut`, and will give them both the value zero. The resulting erratic behaviour of a macro can be very difficult to pin down.

With `Option Explicit`, VBScript will see that you haven't mentioned variables called `TOTAL` and `amonut` in a `Dim` statement, and it will complain and refuse to run the macro: it will even tell you which line the error occurs in. You will then be able to correct your typing mistakes and run a macro that has some chance of success.

We don't use `Option Explicit` in the examples we give you, because it makes them longer without making them any more informative; but in real life `Option Explicit` is a piece of insurance that is well worth considering.

Error handling

VBScript's limited error handling is one of the reasons why it isn't very good for writing large and complex programs: if you want to do this then you should switch to another language – even Visual Basic is better at handling errors.

Let's look again at this macro line:

```
amount=Fields("TOTAL")+0
```

What if the TOTAL field is blank, or contains something that isn't a number? In that case VBScript will report an error and terminate the macro.

Often this is acceptable behaviour, especially when you've got a macro that is working within a single record. You press the button, you see an error message pop up, and you realise that you have forgotten to type something, or have typed something wrong. But displaying an obscure VBScript error message is not the best way to alert a user to the fact that somewhere in his 1,000-record selection there is one record that was badly formatted.

VBScript has just one statement that deals with error handling:

```
On Error Resume Next
```

This makes VBScript ignore the error and carry on to the next statement. The only way you will be able to tell that the error happened at all is to check the built-in VBScript value `Err.Number` – if it is non-zero, an error has occurred.

This behaviour is not a good idea. We have lost count of the number of times that users have asked us for support, saying "I don't know what's going wrong with my macro: I stopped it displaying error messages by adding `On Error Resume Next` but it's still not giving me the right answer". `On Error Resume Next` doesn't correct errors, it only hides them. Don't use it.

The only exception that we consider acceptable is this:

- If you have a small function where only one or two lines can possibly cause an error, you can use `On Error Resume Next` inside that function, and then check explicitly for errors and decide what to do about them. Here's an example:

The On Error Resume Next inside this function doesn't contaminate the rest of your macro, because as soon as VBScript leaves the function it returns error handling to its previous state.

```
Function BulletproofNumericValue(str)
 On Error Resume Next
 BulletproofNumericValue=Trim(str)+0
 If Err.Number<>0 Then BulletproofNumericValue=0
 End Function
```

This function converts a string representing a number (such as "123") into a number, and converts anything else at all ("", or "TWELVE") into zero.

Useful tools for development

Books and other sources of information

This book gives a rapid summary of VBScript in connection with Cardbox. For comprehensive coverage of every VBScript object and function, or for a more graded tutorial, consider the following books, which some of the early testers of Cardbox 3.0 have found useful:

VBScript in a Nutshell: a desktop quick reference - Childs, Lomax, Petrusha (O'Reilly). An excellent book, giving full detail of every command, statement, and function, and describing all the variations of VBScript in Active Server Pages, Outlook, Internet Explorer, and even the Windows Scripting Host. ISBN 0-59600-488-5.

Windows 2000 Scripting Guide (Microsoft Press). Based in part on Microsoft's online reference material. "Comprehensive and covers VBScript in a logical sequence." ISBN 0-7356-1867-4.

Learn VBScript in a Weekend - Ford (Premier Press). "It is aimed at beginner to intermediate. While it is not comprehensive it covers most of the essentials and gives simple examples, which are relatively easy to follow, even though they tend to be web based." ISBN 1-93184-170-5.

Microsoft Script Debugger

When an error occurs, Cardbox reports what the error is and its location in the macro, so that you can find it easily.

You can download a tool called the Microsoft Script Debugger from Microsoft's web site, and so we have included an option in Tools > Options » Macros to start the debugger if an error is detected in a script. Our experience with this debugger has not been encouraging and we have found it better to leave this option turned off.

Displaying information

Especially in the early stages of developing a brand new macro, you may need to see what is going on, to make sure it is working correctly or to investigate why it isn't. There are two ways of displaying useful information:

MsgBox

For a full specification of MsgBox, see page 37.

`MsgBox` displays a message on the screen. You can pass virtually anything to `MsgBox` and it will turn it into a string and display it. For example:

```
MsgBox Fields("TITLE")
```

will display the contents of the field called TITLE in the current record, and

```
MsgBox "Date is " & DateToCardbox(Date+30,"-")
```

will display “Date 1-24-2005” if you run it on 25 December 2004.

`MsgBox` is particularly useful for experimenting with functions that you haven't used before, to check that you understand them before you start relying on them.

WriteLog

To find out the location of your log file, use Help > About Cardbox.

`WriteLog` writes a line to Cardbox's log file. The advantage of `WriteLog` is that you will be able to see in a single file all the messages your macro has generated.

If you're puzzled by a macro's behaviour then insert a lot of `WriteLog` commands into it, to create a detailed history of exactly what it was doing and what the values of all the variables were. Be generous in the amount of logging information written out: large log files are not a problem for Cardbox or for Windows.

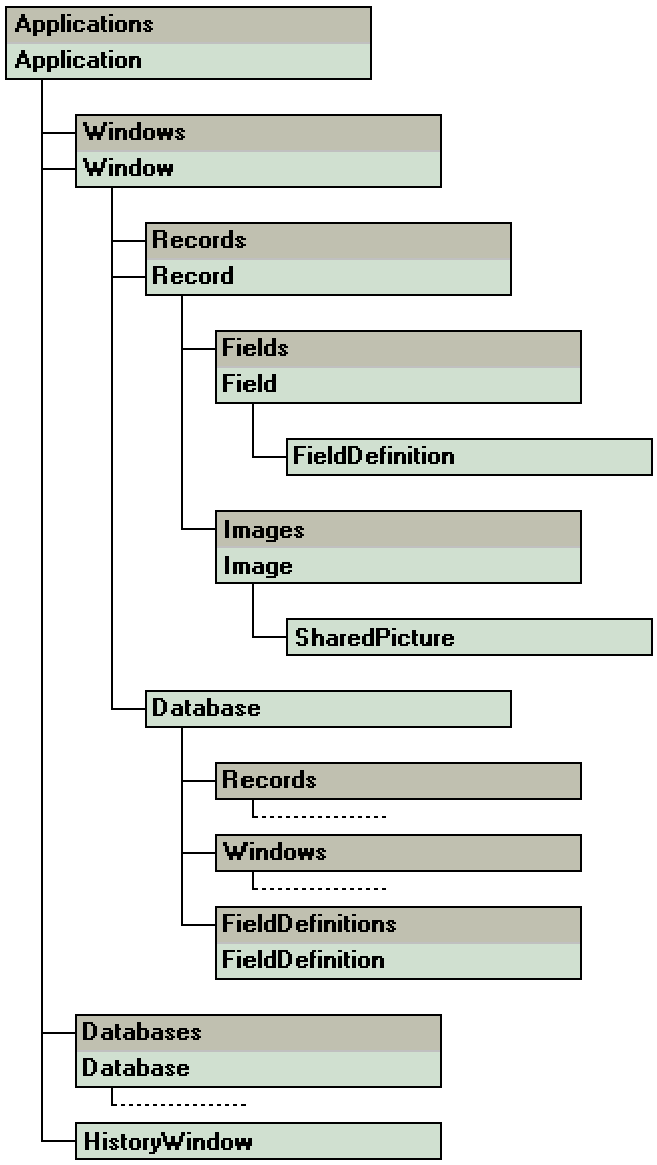
If you have inserted `WriteLog` or `MsgBox` commands and have sorted out your problems, you can comment them out by putting an apostrophe in front of them. For example:

```
'WriteLog "Record " & pos & " title = " & Fields("TITLE")
```

This line now does nothing at all, because it's nothing but a comment; but if a problem occurs then you can quickly convert it back to a working `WriteLog` command just by removing the apostrophe at the beginning of the line.

Part Four
Introduction to objects

The Cardbox object model; converting a macro to use objects.



Applications: p.113
Application: p.113
Databases: p.128
Database: p.128
Fields: p.124
Field: p.125
FieldDefinitions: p.130
FieldDefinition: p.130
HistoryWindow: p.114
Images: p.126
Image: p.127
Records: p.121
Record: p.122
SharedPicture: p.127
Windows: p.115
Window: p.116

Why use objects?

What ordinary macros do

If you record a macro and play it back, it sends a sequence of commands to Cardbox just as you would, except that it types faster. If you enhance the macro by using VBScript, you add to its intelligence but it is basically still doing the same thing as far as Cardbox is concerned: entering commands.

The Cardbox object model lets your macro interact with Cardbox at a deeper level than the normal commands. This can be faster and more direct. To take one example: if you wanted to copy the contents of one field to another, you would have to go to the first field, highlight it all, copy it to the Clipboard, go to the second field, highlight it all, and paste the replacement text from the Clipboard. If you turn on the macro recorder while you're doing this it will record your actions and when you play the macro it will reproduce them faithfully. But the operation is still quite complicated, whereas by using the Field object in Cardbox, you can copy one field to another in one action.

Here are some of the inconveniences of ordinary macros, and how using objects can help:

- An ordinary macro can only touch what is under its nose. To put something into a field, it has to move to that field. To edit a record, it has to move to a record. To select records, it has to make a selection in a window. This destroys some information about where *you* are in Cardbox. If it moves to a different field, you are no longer in the same field. If it moves to a different record, you are no longer on the same record. If it has to make selections of its own, it will probably forget any selections that you have made.
- For the same reason, an ordinary macro also has to put a lot of effort into moving to the right place before it can do something: this housekeeping makes it less easy to see what the macro is meant to be doing.
- When Cardbox gets commands from a macro, it responds to them just as if you had typed them. This means that the toolbar changes when a record is edited; the screen display changes when the macro moves from one record to another; and if the macro switches between one window and another then the whole screen has to change. This is visually distracting but, far more important, it wastes a lot of time. If you are going to process hundreds of records then an ordinary macro will be very slow.

If your macro makes a change to a record that's visible on the screen, the screen display will change to reflect the changes that have been made.

With objects, on the other hand, a macro can refer to any field or any record, whether it's the currently active one or not. It can edit a record without having to issue an Edit command, so that Cardbox doesn't have to change its toolbars, its status bar or anything else. You can think of the macro as interacting with Cardbox "underneath" the normal user interface.

Some examples using the Record object

To get an idea of the difference, here is the tax calculation macro that we created earlier:

See page 23.

```
amount=Fields("TOTAL")+0
net=Round(amount/1.175,2)
GoToField "NET"
TypeText FormatNumber(net,2,True,False,False)
GoToField "VAT"
TypeText FormatNumber(amount-net,2,True,False,False)
```

This moves to a new field, and types text into it, moves to a new field, and types text into it. In the end it leaves you in the VAT field irrespective of where you happened to be when the macro started. (It also, incidentally, has a defect: if the NET or VAT field isn't empty to start with, it will type text in front of what is already in the field instead of replacing it). Here's an equivalent using the Record object:

In a macro, ActiveWindow.ActiveRecord can be abbreviated to ActiveRecord, but we're using the longer version so that you can see more clearly what's going on.

```
Set rec=ActiveWindow.ActiveRecord
amount=rec.Fields("TOTAL")+0
net=Round(amount/1.175,2)
rec.Fields("NET")=FormatNumber(net,2,True,False,False)
rec.Fields("VAT")=FormatNumber(amount-net,2,True,False,False)
```

The differences look small but they are significant. No commands at all are being issued to Cardbox as such. Instead, everything happens through a Record object that represents the currently active record. Your position in the record won't change: only the contents of the NET and VAT fields will be different. And as a bonus, it's clearer what it is that the macro is meant to do.

Let's take this example further by looking at the batch version of the tax calculation. To remind you, here it is:

See page 24.

```
For pos=1 to RecordCount
GoToRecord pos
EditRecord

    amount=Fields("TOTAL")+0
```

```
        net=Round(amount/1.175,2)
        GoToField "NET"
        TypeText FormatNumber(net,2,True,False,False)
        GoToField "VAT"
        TypeText FormatNumber(amount-net,2,True,False,False)
    SaveRecord
    Next
```

This steps through every record in the current selection – if you play a macro like this, you'll see it happening – and it leaves you on the last record, irrespective of where you may have been when you started playing the macro. So there's a lot of screen activity, and at the end of it all your original position is lost.

Here is the version that uses objects:

In a macro, ActiveWindow.Records can be abbreviated to Records, but we're using the longer version so that you can see more clearly what's going on.

```
For Each rec In ActiveWindow.Records
rec.Edit
  amt=rec.Fields("TOTAL")+0
  net=Round(amt/1.175,2)
  rec.Fields("NET")=FormatNumber(net,2,True,False,False)
  rec.Fields("VAT")=FormatNumber(amt-net,2,True,False,False)
rec.Save
Next
```

Again, there are no real Cardbox commands in this, and the macro uses the Record and Records objects to do all its work. If you watch this macro, you won't see anything happening except at the moment when its processing goes through the records that are currently on the screen; and even then, all that happens is that the NET and VAT fields change to reflect the new values they have just been given. Your current position in the selection won't be affected, and the whole thing will go much faster than the previous version.

This macro refers to the state of affairs on the screen: look on the first line, where `ActiveWindow.Records` creates a Records object that represents all records in the current selection. It's quite common to have this kind of reference at the start of a macro, but it's not essential: if you used

```
For Each rec In ActiveWindow.Database.AllRecords
```

instead, then the macro would process every record in the whole of the database.

You can see this macro on page 102.

A final example: one Cardbox user has a database of invoices, and because he has multiple shops, he wants to be able to fill in the shop's address and telephone number automatically when he creates an invoice. So he enters a code in one field of his database, and he has a macro that fills in the other information once he's done this. The macro searches a separate Shop Addresses database, finds a record corresponding to the code he entered, and extracts the address and telephone number from it. By using objects, the macro can do this without having to switch windows, move among fields, or anything else: this is faster and more convenient. The user doesn't even have to know that the Shop Addresses window exists.

The Cardbox object model

You can see a chart of all Cardbox objects and their interrelations on page 58.

A detailed description of all Cardbox objects starts on page 109.

Cardbox makes its internal organisation visible to VBScript (and other programming languages) as objects of various types. The most important object types are Window, Records, Record and Field.

Window represents a database window within Cardbox. When you record a macro, it is mostly composed of a list of commands to be sent to a Window object (the Window object that represents the currently active window).

Records represents a selection of records. A good thing about Records objects is that you can use them to perform searches and selections without affecting the current selection on the screen.

Record represents a record in a database. It can represent any record at all, not just the one that is currently active.

Field represents the contents of a single field in a record. If you want to retrieve or change the contents of a field, the Field object is the fastest and most direct way of doing it.

The Application object

See page 113 and Help Point 721.

The Application object represents a copy of Cardbox that is currently running. You can use it to resize or move the Cardbox window and to access the Windows Clipboard.

Example: copying text to the Clipboard

```
    ClipboardText=Fields("AD")
```

will take the text of the field named AD in the active record in the active window and copy it to the Clipboard

Example: closing Cardbox

To close Cardbox from a macro, use

```
Application.Visible=False
```

Cardbox won't actually close down until the macro has finished, so you would usually make this the last command in your macro.

See page 115 and Help Point 723.

The Windows object

The Windows object is a collection of Window objects representing database windows within Cardbox. Normally it will represent all the windows open within Cardbox, but you can also use it to find all the windows that have a specific database in them.

OpenFile has a number of options, which are listed in Help Point 751; or you can turn on the macro recorder, open a database, and then look at what gets recorded.

Example: opening a database

```
Set win=Windows.OpenFile("C:\MyData\Sample File.fil")
```

opens the named database and stores a Window object for the newly opened window in the variable `win`.

```
Windows.OpenFile("C:\MyData\Sample File.fil")
```

simply opens the database without storing any reference to the Window object. Since the new database window will always be the active one, you can refer to it with `ActiveWindow` if you need to.

See page 116 and Help Point 724.

Exception: you have to say ActiveWindow.Left and ActiveWindow.Select, because Left and Select are keywords built into Visual Basic.

The Window object

Almost any menu command that you enter into Cardbox is effectively an action applied to the currently active window, so the Window object has an enormous number of methods and properties. For this reason the Cardbox macro system provides a short cut: if you want to use a method or property of the active window's Window object, you don't need to specify an object reference at all: so instead of

```
ActiveWindow.GoToRecord 10
```

you can say

```
GoToRecord 10
```

Getting a Window object

```
    Set win=ActiveWindow
```

gets the Window object for the currently active window and stores it in `win`.

```
    Set win=Windows("DIARY")
```

gets the Window object for the window named DIARY: if there isn't such a window, VBScript will report an error. (The point of getting a Window object for a window other than the current one is that you can then access that window's records and even send commands to it without having to switch back and forth between windows on the screen).

Making selections

Each of the possible search commands in the Search menu has a corresponding method in the Window object.

It is also possible to use the Records object to do selections directly without involving the database window at all: this can be faster and less obtrusive.

The Records object

See page 121 and Help Point 725.

The Records object represents a selection of records: it is a collection of Record objects representing records in a database. The Records object lets you do a few specific things such as writing out or printing selected records, tagging or untagging them; and it is also useful for making searches and selections without having any effect on the selection that the user sees on the screen.

Getting a Records object

`Records`	The current selection in the active window.
windowObject`.Records`	The current selection in the given window.
databaseObject`.AllRecords`	All the records in the database.
databaseObject`.NoRecords`	An empty selection.
databaseObject`.TaggedRecords`	The tagged records in the database.

If you change levels of selection, or create or delete records, after getting a Records object, the Records object won't be altered to reflect the changes.

Adding or removing records

Add and Remove don't create or delete records: they only affect their presence in a selection.

The `Add` and `Remove` methods let you add records to the selection or remove them. They can be useful if you are trying to do a kind of "intelligent tagging" – looking through the database to see which records need to be selected – without having to use the normal tagging mechanism.

Making selections

The Records object has a comprehensive set of search methods that mirror the Search menu and the methods of the Windows object. There is a difference in behaviour, though: search methods in the Windows object alter the current selection in the window, but the search methods of Records don't affect the Records object they're used on – instead, they create and return a new Records object that represents the result of the search.

Example: making a single selection

You already know how to select records in Cardbox, using Search > Clear to get to Level 0 and then Search > Select to select records matching your chosen criterion. Here is the equivalent with Records objects.

First, get a Records object that contains all the records in the database:

```
Set recsAll=Database.AllRecords
```

Next, apply the Select method to get the selection you want:

```
Set recsSelected=recsAll.Select("field","value")
```

You can then use `recsSelected` in whatever way you want: for example, you can export records from it to a file, or you can go through the records and edit them, or you can use it as the basis of another selection.

Use whichever idiom is most convenient, or whichever one you think you'll be likeliest to understand when you look at your macro again in a few months' time.

You could do the same selection in one step:

```
Set recsSelected=Database.AllRecords.Select("field","value")
```

Or you could use a single variable, like this:

```
Set recs=Database.AllRecords
Set recs=recs.Select("field","value")
```

This works because the Select method creates a new Records object, and this is then stored in `recs` and replaces the Records object that was originally there.

Example: multiple selections

You could imagine that window A holds customers and window B holds product details, but the principles illustrated are quite general.

Suppose that you have two databases in two windows called A and B. You have to select, in window B, just those records that match any record in window A. This is like a relational search, only more so: a relational search could find you the records in B that matched *one* record in A, but here you want the records that match *any* of them. Here is one way that you might do this. We'll assume that the fields to be matched are called FLDA in database A and FLDB in database B.

The line numbers aren't part of the macro: they are there to make the commentary easier to follow.

```
1 Set recsB=Windows("B").Records
2 Set recsFound=Windows("B").Database.NoRecords
3 For Each rec in Windows("A").Records
4  Set recsTemp=recsB.Select("FLDB",rec.Fields("FLDA"))
5  Set recsFound=recsFound.IncludeFromRecords(recsTemp)
6  Next
7 Windows("B").SelectFromRecords recsFound
```

1. `recsB` is the current selection of records in window B. We'll be using this selection repeatedly, so it is a lot more efficient to retrieve it once and store it in a variable.
2. `recsFound` is a collection of the records that have been found to match so far. At the end of the process we'll use it to perform a selection in window B.
3. `For Each` has the effect of repeating lines 4 to 5 once for each record in the current selection in window A, with the variable `rec` containing that record's Record object.

Lines 4 and 5 could be combined, since recsTemp isn't used elsewhere, but the macro wouldn't be any faster and it would be rather less readable.

4. We take the value of field FLDA in `rec`, and feed it into a Select command that operates on `recsB`. Thus `recsFound` ends up being the result of a single Select command on the current selection in window B.
5. We combine `recsTemp` with `recsFound` and store the result back into `recsFound`. As the loop progresses, `recsFound` will steadily grow until it contains all the records we need.
6. `Next` marks the end of the `For Each` loop.
7. Now `recsFound` contains all the records we want, but it has to be made visible to the user somehow. We could use either `SetFromRecords` or `SelectFromRecords` to alter the current selection in window B. On the whole, `SelectFromRecords` seems better, because that will behave like a single Select command from the user's point of view. This means that the whole macro will feel like a single Select command, and so can easily be undone by the user when the selection is no longer needed.

To do the same kind of thing without using objects, you'd use kept selections and the Keep > Keep command.

The Record object

See page 122 and Help Point 726.

The Record object represents a record in a database. You can use it do things to the record, such as tagging, editing and deletion, and to get hold of a Fields object that gives you access to the individual fields.

In addition, the For Each statement applied to a Records object can be used to get a Record object for each record in the corresponding selection.

Getting a Record object

`ActiveRecord`	The current record in the active window.
windowObject`.ActiveRecord`	The current record in the given window.
databaseObject`.NewRecord`	A newly created record, ready for editing.
recordObject`.DuplicateRecord`	A newly created duplicate of an existing record, ready for editing.

How editing works

One way of editing records is to use the `EditRecord` and `SaveRecord` methods of the Window object. These act on the currently active record, and they are the exact equivalent of the user entering the Edit > Edit Record and File > Save commands.

- These are the methods that are stored in a macro if you turn macro recording on and then start editing a record.
- While you are editing in this way, you can use methods of the Window object such as `TypeText` and `GoToRecord`. Alternatively, you can change the contents of a field by obtaining a Field object and setting its `Text` property directly.
- You can only edit the current record, you can only edit one record at a time in any given window, and the Cardbox user interface (menus, toolbar, status bar) will change appearance when you start and stop editing a record.

The alternative is to use the `Edit` and `Save` methods of the Record object.

- These methods operate independently of any Window object.
- You can edit any record you like, and as many records as you like.
- While you are editing, you can change the contents of a field by obtaining a Field object and setting its `Text` property directly.
- Nothing happens to the screen when records are edited and saved (except that if a record is visible on the screen then it will be updated to show any changes that were made).

If the user is currently editing this record

If the user is editing a record while your macro makes changes to it, then if the user discards all changes using File > Quit Without Saving, the changes your macro made will be discarded as well. In some cases this is a bad thing (for example, if your macro is editing a whole group of records and you want the result to be uniform) but in other cases it doesn't matter. When you use the `Edit` method you can choose whether it should be an error for `Edit` to be applied to a record that is being edited by the user. Your macro could also use the `UserEditing` property to detect whether the user was editing a record and take special action in that case.

The Fields object

See page 124 and Help Point 727.

The Fields object is not very interesting. Its principal use is as a way of getting hold of the Field objects that represent individual fields.

Getting a Fields object

`Fields`	The fields in the current record in the active window.
recordObject`.Fields`	The fields in the given record.

The Field object

See page 125 and Help Point 728.

The Field object represents the contents of a field. Getting its Text property retrieves the contents of the field and setting its Text property changes the contents of the field.

Getting a Field object

`Fields("`*name*`")`	The field called *name* in the current record in the active window.
recordObject`.Fields("`*name*`")`	The field called *name* in the given record.
fieldsObject`("`*name*`")`	The field called *name* in the given Fields object.

Using the Text property

When retrieving text

The official way to retrieve the text of a field is to use its `Text` property: for example,

```
MsgBox "The field is " & Fields("TEL").Text
```

or

```
Set fld=Fields("TEL")
MsgBox "The field is " & fld.Text
```

Because `Text` is the “default” property of the Field object, you can use the following abbreviated forms:

```
MsgBox "The field is " & Fields("TEL")
```

or

```
Set fld=Fields("TEL")
MsgBox "The field is " & fld
```

and VBScript will assume the `Text` property automatically.

When setting text

The official way to set the text of a field is to use its `Text` property: for example,

```
Fields("TEL").Text="555-1212"
```

or

```
Set fld=Fields("TEL")
fld.Text="555-1212"
```

With the first form, abbreviation is possible:

```
Fields("TEL")="555-1212"
```

But don’t do this:

```
Set fld=Fields("TEL")
fld="555-1212"
```

won’t do what you expect, because the second line will just tell VBScript to discard the existing reference to a Field object and store a string value directly into the variable called `fld`. So this is one case where you really do have to use the `Text` property.

Indexed words and the TextFormat property

Strings in VBScript don't have any special index markers, which means that when you use the `Text` property to get text from a Cardbox field into a VBScript variable, all the indexing information is lost: if you have a field called CO containing the words "**Smith** & Son", where Smith is indexed and the rest of the field isn't, doing `txt=Fields("CO").Text` will result in `txt` containing the string `"Smith & Son"`, with no indication of what was and wasn't indexed.

This is often a good thing, since a lot of the text processing that you'll find yourself doing in VBScript doesn't pay attention to indexing – but sometimes you do need to know about indexing. One obvious and common case is when you're copying text from one field into another, and some words are indexed while others aren't.

Copying from one field to another

By default, the text will be indexed if the field's indexing mode is All or Auto and not indexed if the mode is Manual or None.

The problem with the `Text` property is that extracting text from a field into a string using `Text` strips all indexing information from it. Storing this plain text into a field then indexes it according to the field's indexing mode.

If this is good enough for you, you don't need to read any further: you can use any method you like for transferring text between fields.

The best and easiest way to copy text from one field to another is:

```
Fields("TO") = Fields("FROM")
```

You can copy several fields at once using this technique: see page 99.

If you are able to use this then you also don't need to read any further – except, perhaps, to see why it's so good:

1. `Fields("FROM")` results in a Field object (not a string of text).
2. Storing this into `Fields("TO")` tells Cardbox to store the contents of one Field object into another.
3. Cardbox is intelligent enough to understand that this means that you want to copy indexing information from one field to another, not just the text.

Note that this only works when you're copying fields directly. If you use something like `x=Fields("FROM") : Fields("TO")=x` then VBScript will convert the field to a string on the way (to store it into the variable `x`), which will lose all the indexing information. For the same reason, this simple method won't work if you want to do something cleverer than just replacing the whole of the destination field with the entire contents of the source field.

The TextWithIndex property

TextWithIndex behaves a lot like Text but it uses special markers to indicate indexing. You've already seen that for a field containing "**Smith** & Son", Text gives "Smith & Son". TextWithIndex takes account of indexing, so it represents this field as "*Smith & Son".

TextWithIndex is a property that can be both read and written, so here is another way of copying data from one field to another while preserving indexing information:

```
Fields("TO").TextWithIndex = Fields("FROM").TextWithIndex
```

Here is how it works:

You can still have asterisks in your field text if you want: the TextWithIndex format is designed to work with all possible field contents.

1. Fields("FROM").TextWithIndex results in a string representing the contents of FROM, with indexed words marked by asterisks.
2. Storing this into Fields("TO").TextWithIndex makes Cardbox interpret those asterisks as index markers.

What this means is that indexing information won't be lost while you're processing field text as VBScript strings. Here, for example, is how to add text from one field to the end of another field:

```
textTo=Fields("TO").TextWithIndex
textAdd=Fields("FROM").TextWithIndex
if Len(textTo)=0 Then
  Fields("TO").TextWithIndex=textAdd
 Else
  Fields("TO").TextWithIndex=textTo & vbLf & textAdd
 End If
```

In detail, the macro checks whether the TO field is empty. If it is, it just copies FROM into it; but if TO already contains some text, it combines the original text with the contents of FROM (inserting a new-line marker in between, so that the contents of FROM will start on a separate line) and stores it into TO. Because the macro uses TextWithIndex, the original indexing state of the text being processed will be unharmed by all this string manipulation: indexed words will stay indexed, and unindexed words will stay unindexed.

The TextFormat property

An alternative to remembering to use `TextWithIndex` every time is to use `TextFormat` to change the meaning of the `Text` property itself.

For more details of these text formats, see Help Point 754.

Let's consider the old example again: a field containing the words "**Smith** & Son", where "Smith" is indexed and "& Son" isn't. The value of the Field object's `Text` property will depend on the object's `TextFormat` property, as follows:

The TextLength property will always be 11, because the field is 11 characters long.

TextFormat property	Text property
0 (`cbxTextNormal`)	`Smith & Son`
1 (`cbxTextIndex`)	`*Smith & Son`
2 (`cbxTextXML`)	`Smith &amp; Son`
3 (`cbxTextXMLIndex`)	`<x>Smith</x> &amp; Son`

If you don't explicitly set TextFormat anywhere, the default is cbxTextNormal.

These different formats are useful for different purposes. For straightforward display and processing, `cbxTextNormal` is best; `cbxTextIndex` lets your macro see what words are indexed; the XML formats are better if the programs you are going to communicate with also understand XML.

Using the `Text` property with `TextFormat` set to `cbxTextIndex` is exactly the same as using `TextWithIndex`. Which of the two approaches you use is up to you: do whatever seems most convenient and least prone to errors.

There are two ways of setting the text format. One is to set the `TextFormat` property of the Fields object, in which case all Field objects that are created from that object will receive that text format. The other is to set the `TextFormat` property of a Field object before you use it to retrieve or set the text of the field.

Object lifetimes: a warning

You might think that the most natural way of using `TextFormat` should be something like this:

```
Fields("NAME").TextFormat=cbxTextIndex
MsgBox "The text is " & Fields("NAME")
```

Unfortunately this won't work. The first line will create a Field object and set its `TextFormat` property, but because you haven't saved the object into a variable, VBScript will discard the newly created Field object, and the second line will create a brand new Field object that will not remember the `TextFormat` you gave its predecessor. The correct approach is something like this:

```
Set fld=Fields("NAME")
fld.TextFormat=cbxTextIndex
MsgBox "The text is " & fld
```

You could try to save time by using the `TextFormat` property of the Fields object, but again you'll have to be careful, because `Fields` isn't a Fields object itself, just a property that creates a Fields object for the current record. Using `Fields` twice will create two successive Fields objects, and the text format of the second one won't be affected by the `TextFormat` setting that you used in the first. So if you do decide to use the Fields object, the alternative should end up looking like this:

You can also set the TextFormat property of a Record object, to control the TextFormat property of all the Fields objects created from it (and all their Field objects).

```
Set flds=Fields
flds.TextFormat=cbxTextIndex
MsgBox "The text is " & flds("NAME")
```

This approach is no shorter than the others if just one field is involved, but if you are processing several fields then it saves a lot of space, since setting the `TextFormat` property for the Fields object means that you don't have to set the property separately for every field.

The Images object

See page 126 and Help Point 729.

The Images object represents the images in a record. You can use it to get hold of an Image object for any particular image, or you can use its methods to add new images, from files or by pasting them from the Clipboard. (There is no method for scanning images because the interaction with TWAIN, and with scanner drivers generally, makes scripting impossible).

Getting an Images object

`Images`	The images in the current record in the active window.
recordObject`.Images`	The images in the given record.

The Image object

See page 127 and Help Point 730.

The Image object represents an image.

Getting an Image object

`Images(`*n*`)`	The *n*th image in the current record.
recordObject`.Images(`*n*`)`	The *n*th image in the given record.
imagesObject`(`*n*`)`	The *n*th image in this image object.
imagesObject`.ReadFromFile`	Reads an image from a file and returns an Image object that represents that image.
imagesObject`.Paste`	Pastes an image from the Clipboard and returns an Image object.

Example: importing an image from a file

```
    Set img=Images.ReadFromFile "C:\MYFILES\TESTIMG.JPG"
```

reads an image from a file. Until the record containing the image is saved, you can use properties of the Image object to rotate the image and control its size and the degree of compression to be used.

The Databases object

See page 128 and Help Point 732.

The Databases object isn’t much used, but it is a way of getting a list of all the databases open in this particular copy of Cardbox.

Getting a Databases object

`Databases`	The Databases object for this copy of Cardbox.

The Database object

See page 128 and Help Point 733.

The Database object represents a Cardbox database. It lets you do things that affect the database as a whole, such as removing tags or deleting kept selections. It also lets you create Records objects referring to the database.

In addition, the For Each statement applied to a Databases object can be used to get a Database object for each database currently open in this copy of Cardbox.

Getting a Database object

`Database`	The database open in the active window.
windowObject`.Database`	The database open in the given window.

Example: automatic backup

Suppose that you have a workspace containing a number of databases that reside on a remote server. Then you can use the following macro to download backup copies of them all onto your PC:

```
For Each db In Databases
 DownloadOneDatabase db
 Next

Sub DownloadOneDatabase(db)
 filename=db.FullName
 If Left(filename,10)<>"cardbox://" Then Exit Sub
 posName=InstrRev(filename,"/")
 fname="C:\Downloads\" & Mid(filename,posName+1)
 db.Download fname & ".FIL",cbxDownloadFIL
 db.Download fname & ".FMT",cbxDownloadFMT
 End Sub
```

As a refinement, you could modify this macro to look at the name of the server and use it to decide which folder the database should be downloaded into.

The For Each / Next loop calls the subroutine once for every database that is currently open. The subroutine looks at the filename of the database. If the filename doesn't start with “cardbox://” then the database is not a server-based one but is on your local hard disk, so it can't be downloaded and the subroutine exits. Next, the subroutine finds the last slash in the filename, because this separates the server name from the database name. Then it constructs a filename on your hard disk based on that database name (obviously, you could use a different folder to store the downloaded copy of the file). Finally it downloads both the database file and the format file, using the appropriate filetype in each case.

Other Cardbox object types

Applications (Help Point 720) lists all the currently open copies of Cardbox: you would never use it in a macro, but it could be useful if you had a separate program that wanted to connect to Cardbox but needed to ask you which copy to connect to.

HistoryWindow (Help Point 722 and page 114) lets you control the size and position of the History of Selections window.

FieldDefinition (Help Point 735 and page 130) tells you the index mode and other attributes of a field.

Connection (Help Point 736 and page 131) lets you open a TCP/IP connection from within a macro and send and receive data on it. It could be used, for example, as part of a macro that sent email directly from Cardbox without having to depend on a separate email program.

Part Five

Objects outside Cardbox

FileSystemObject; sending faxes; some approaches to email; controlling Microsoft Office.

VBScript can create objects of any type as long as the software for that object type is currently available on your Windows system. We've documented the FileSystemObject object because it's important and it's built in to every Windows system, and we've documented a few other objects to give you an idea of the principles involved. In general the rule is always the same: find the documentation for the program that that you want to use, and read it. Failing that, contact the manufacturer and ask whether the program has a scripting interface that can be used with VBScript.

FileSystemObject

For someone coming to VBScript from another programming language, one of the most puzzling features is the lack of any built-in commands for reading or writing files. But although the language doesn't handle files, an object is provided for file access, called FileSystemObject.

Getting a FileSystemObject

```
    Set fso=CreateObject("Scripting.FileSystemObject")
```

creates a FileSystemObject and stores a reference to it in the variable `fso`.

Opening and reading a file

For other options of OpenTextFile, see Microsoft's documentation on the Web, or one of the books listed on page 55.

```
    Set stm=fso.OpenTextFile("filename")
```

opens a file for reading, creates a TextStream object and stores a reference to it in the variable `stm`. The TextStream object has a number of methods and properties, of which the following are the most important:

`AtEndOfStream`	`True` if the end of the input data has been reached, `False` if there is still something left to read.
`Read(`*n*`)`	Reads *n* characters and returns them as a string.
`ReadLine`	Reads an entire line of text and returns it as a string.
`ReadAll`	Reads the whole file and returns its contents as a string.

After you have finished reading a file, you should close it using the `Close` method (after Close, you can't use any other methods of the TextStream object).

Creating and writing a file

```
Set stm=fso.CreateTextFile("filename")
```

creates a file, opens it for writing, creates a TextStream object and stores a reference to it in the variable stm. If a file with the chosen name already exists, VBScript will report an error and will not overwrite the file.

```
Set stm=fso.CreateTextFile("filename",True)
```

is identical except that if the file already exists, it will be overwritten and no error will be reported.

The TextStream methods for writing to a file are:

`Write "text"`	Writes the given text to the file.
`WriteLine "text"`	Writes the given text to the file and then starts a new line.

After you have finished writing a file, you should close it using the `Close` method.

Example: translating a file format

Suppose that you have data that you want to import into Cardbox, but the data file has the layout of a printed report, with vertical bars separating the columns:

```
1  |2      |3|4
```

The aim is to turn this into a comma-separated format that Cardbox can read easily:

```
"1  ","2      ","3","4     "
```

Here is one way of doing this:

```
Set fso=CreateObject("Scripting.FileSystemObject")
Set input=fso.OpenTextFile("InputFile.txt")
Set output=fso.CreateTextFile("OutputFile.txt")
While Not input.AtEndOfStream
 fullLine=input.ReadLine
 items=Split(fullLine,”|”)
 For i=LBound(items) To UBound(items)
  items(i) = """" & items(i) & """"
  Next
 output.WriteLine Join(items,",")
 Wend
input.Close
output.Close
```

We're not going to give a detailed commentary on this macro, but you should be able to see how the `ReadLine` and `WriteLine` functions are used to read from one file and write to the other.

- If you are faced with a file with odd separator characters, check the documentation for the `ReadFromFile` method before you start, because you can use it to specify separator characters other than the comma: so you may be able to get away without doing any conversion at all if the vertical bars were your only problem.
- The output file contains spaces at the end of each field because we didn't feel like removing them, and Cardbox removes trailing spaces anyway when it imports a file. If you want to remove spaces yourself during the conversion, replace the line inside the innermost loop with the following:

```
items(i) = “””” & Trim(items(i)) & “”””
```

Other functions

FileSystemObject contains methods and properties that copy, rename, list, create and delete files and folders. For full details see one of the reference books listed on page 55 or see Microsoft's own documentation on the Web.

Sending faxes

Like most Windows programs, Cardbox can print documents but it can't send them as faxes: the closest it can get to this is "printing" to a fax device which then pops up a window so that you can fill in the fax number to be used.

Scripting objects let you tell the fax system where to send the fax, so that no window pops up and it's possible to send bulk faxes without human intervention. We give examples for Symantec's WinFax Pro and one for Microsoft Fax Services for Windows XP, because their architectures are different and it's useful to see how a script can cope with the differences. Help Point 512 gives more options and details.

WinFax Pro

Although the WinFax scripting interface has remained the same for years, Symantec may change it at any time: so please consult their latest documentation before trying to send faxes.

WinFax Pro from Symantec has an object interface that allows you to send faxes directly from other programs including Cardbox.

Sending standard faxes

In this scenario, you have a document file that contains a fax you would like to send, and your current selection lists the people you would like to send this fax to.

We assume that a field called RECNAME contains a short name of the recipient, to make WinFax's status reports more meaningful. If you haven't got a suitable field, remove all references to RECNAME and NAMEFIELD.

```
Const NUMBERFIELD = "FAXNO"
Const NAMEFIELD = "RECNAME"

Set winfax=CreateObject("WINFAX.SDKSEND8.0")
winfax.ShowSendScreen(0)
winfax.AddAttachmentFile("c:\Samples\Message.Doc")
For Each rec in Records
  winfax.SetNumber(rec.Fields(NUMBERFIELD))
  winfax.SetTo(rec.Fields(NAMEFIELD))
  winfax.AddRecipient()
  Next

winfax.Send(1)

' Wait for WinFax to finish queuing the fax.
Do While winfax.IsEntryIDReady(0) <> 1
  Sleep 20
  Loop

winfax.Done
```

- We use `Const` declarations to isolate the field names and make them easier to change.

- We have copied all the WinFax commands straight from the WinFax documentation.
- Having started a fax job by giving WinFax an “attachment file”, we loop through every record in the current selection. Each `SetNumber` / `SetTo` / `AddRecipient` cycle adds a new recipient to WinFax’s list.
- `Send` tells WinFax to start adding the new fax to its queue of outgoing faxes. We then wait in a `Do While` loop until WinFax reports that it has accepted the queued fax request.
- Finally, we use WinFax’s `Done` method, because the documentation says that we must: it implies that something will go wrong if we don’t do this, although it doesn’t say what.

Sending customised faxes

You may prefer to send a different fax to each of your recipients. In that case, create a format that will print the fax in exactly the form you want the recipient to see it, and use View > Change Format to make that the active format.

```
const NUMBERFIELD = "FAXNO"
const NAMEFIELD = "RECNAME"

Set winfax=CreateObject("WINFAX.SDKSEND8.0")
winfax.SetPrintFromApp(1)
winfax.ShowSendScreen(0)

Set recs=ActiveWindow.Records
n=recs.Count

For i=1 to n
  Set rec=recs(i)
  winfax.SetNumber(rec.Fields(NUMBERFIELD))
  winfax.SetTo(rec.Fields(NAMEFIELD))
  winfax.AddRecipient()
  winfax.Send(1)
  Do While winfax.IsReadyToPrint=0
    Sleep 20
    Loop
  Print cbxPrintMainRecord,,i,i,"WinFax"
  Do While winfax.IsEntryIDReady(0) <> 1
    Sleep 20
    Loop
  Next

winfax.Done
```

- The `SetPrintFromApp` method tells WinFax to accept input from a program instead of printing a document file.

- We loop through every record in the current selection. We do this with `For` rather than `For Each` because we need to know our position in the selection, for the `Print` method.
- After the usual `SetNumber` / `SetTo` / `AddRecipient` cycle adds a new recipient to WinFax's list, we use `Do While` to wait until WinFax reports that it is ready to accept printed output.
- Once WinFax is ready, we use Cardbox's `Print` method to print one record to a printer called "WinFax".

There are a great many options that you can set when sending faxes through WinFax but you will need to read the WinFax documentation to find out what they are and how to use them.

Microsoft Fax Services

Sending customised faxes

As before, prepare a format that presents your records in the form that you'd like the recipients to see them, and switch to that format. The macro looks like this:

```
Const NUMBERFIELD = "FAXNO"
Const NAMEFIELD = "RECNAME"
Set server=CreateObject("FaxComEx.FaxServer")
server.Connect ""

Set fso=CreateObject("Scripting.FileSystemObject")
tempFilename=fso.GetSpecialFolder(2) & "\"  _
                                 & fso.GetTempName & ".tif"
Set recs=ActiveWindow.Records
n=recs.Count

For i=1 to n
  Set rec=recs(i)
  Print cbxPrintMainRecord,,i,i,"Fax," & tempFilename
  Set doc=CreateObject("FaxComEx.FaxDocument")
  doc.Body=tempFilename
  doc.Recipients.Add Fields(NUMBERFIELD),Fields(NAMEFIELD)
  doc.connectedSubmit server
  doc=0
  fso.DeleteFile tempFilename
  Next
```

The strategy is different from the WinFax case. With Microsoft Fax, we have to write each fax out as a file and then ask the fax system to send it.

- There are two objects required for sending faxes: a FaxServer object and a FaxDocument object. You only need one FaxServer object, so we create it at the very start of the macro.
- We start by getting hold of a FileSystemObject and using its `GetSpecialFolder` and `GetTempName` methods to create a unique filename.
- For each record in turn, we print the record to a printer whose name is "Fax" combined with the filename we've just chosen. This has the effect of creating a file that is the graphical equivalent of the merged record, in a format suitable for faxing.
- We then create a FaxDocument object and set its "body" (what actually gets faxed) and its recipient. We then use its `connectedSubmit` method to submit the document to the FaxServer that we created earlier.
- Finally, we release the FaxDocument reference by setting the `doc` variable to something else (the number zero was a simple and readable choice), and we delete the temporary file.

There are many options available, covering such things as cover pages and scheduling. Help Point 512 links to a page that lists the options. The same Help Point tells you how to send a single document as a fax to multiple recipients, and even how to send a fax that is just a cover page with a message and has no attached document at all.

Sending emails

Help Point 529 has a comprehensive set of sample macros for MAPI, Eudora, Outlook, or Outlook Express. The principles behind these are essentially the same as those you've seen in the fax examples and it isn't useful to print them all here. The macros don't actually send emails but add them to your email outbox, which lets you use all your email system's facilities for actually transmitting the emails and checking on their progress.

Controlling Microsoft Office

Microsoft Office is an immense product: the Word 2000 Developers' Handbook alone is 1,200 pages long. This means that we can't go into any depth at all with describing how Office (or even Word) can be programmed. This is, after all, a Cardbox book and not a Word one. But just so that you can see that Cardbox can drive Word quite simply, here is a sample macro:

```
Records.WriteToFile cbxWriteFormatCSV+ _
      cbxWriteOptionFieldHeading+cbxWriteOptionAlwaysQuote, _
      "C:\temp.txt"
Set doc=GetObject("C:\merge.doc")
Set mrg=doc.MailMerge
 mrg.Destination=0
 mrg.SuppressBlankLines=True
 mrg.DataSource.FirstRecord=1
 mrg.DataSource.LastRecord=-16
 mrg.Execute False
doc.Application.Visible=True
```

This macro presupposes the existence of a Microsoft Word merge document called `merge.doc` that has been set up to use the file `temp.txt` as its data source.

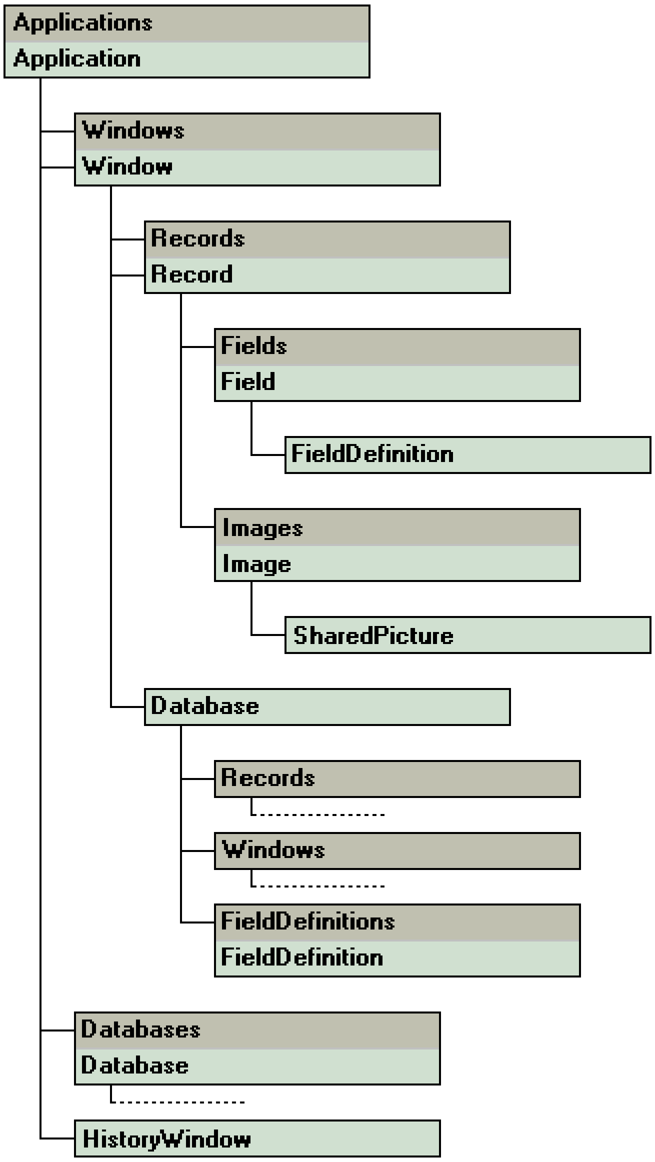
- The macro uses Cardbox's `WriteToFile` method to write selected records to the temporary file. This is the only Cardbox method call in the entire macro.
- The macro then uses `GetObject` to retrieve a Word document object that refers to `merge.doc`, sets various attributes to control the mail-merge operation, executes the merge, and finally makes the document visible so you can see and print the result.

For hints on translating Microsoft Office macros into VBScript, see Help Point 763.

We created this piece of VBScript by using Word's macro recorder to record the action of performing a mail-merge. We then looked at the resulting program code in Word and translated it into VBScript.

Part Six
Using other languages

Visual Basic; VBA in Microsoft Office; VBScript in other contexts; other languages.


Applications
Application
Windows
Window
Records
Record
Fields
Field
FieldDefinition
Images
Image
SharedPicture
Database
Records
Windows
FieldDefinitions
FieldDefinition
Databases
Database
HistoryWindow

Now the boot is on the other foot. Instead of getting Cardbox macros to control other programs, we'll look at getting other programs to control Cardbox.

There are several reasons why you might want to do this. You might decide that VBScript is not a rich enough language for what you want to do, or you might have your own favourite language and feel happiest programming in that. You might also be planning to use Cardbox in conjunction with Microsoft Office and want all the macros to be launched from Office rather than from Cardbox.

Controlling Cardbox from outside is easier than it looks. All the Cardbox objects, methods and properties are accessible from anywhere, and the fact that we've been showing them to you in the context of VBScript running inside Cardbox itself was just a matter of convenience. Every modern language has its own way of accessing objects and their methods and properties, so whether you are using Visual Basic, VBA, Java, Delphi or C++, the knowledge of Cardbox objects that you've acquired so far is still relevant.

Getting a Cardbox object

For more details, and more languages, see Help Point 764.

The key Cardbox object is Application, which represents a running copy of Cardbox. In Visual Basic and related languages, you can get hold of an Application object for an already running copy of Cardbox like this:

```
Set cbx=GetObject(,"Cardbox.Application")
```

You can also create an Applications object and use that to enumerate all the running copies of Cardbox until you find the one you want: for more about this, see Help Point 720.

If there is more than one copy of Cardbox running then you don't have any control over which copy your Application object represents. The alternative is to use the filename of the workspace to specify which copy you want to talk to:

```
Set cbx=GetObject("C:\path\filename.cbw")
```

To start a new copy of Cardbox, use this:

```
Set cbx=CreateObject("Cardbox.Application")
```

The newly created copy of Cardbox will be invisible unless you make it visible by using the `Visible` method of the Application object. If you release the Application object (or your program ends) without making Cardbox visible, the copy of Cardbox you have created will silently close itself.

Navigating through Cardbox

The lines on the map of the Cardbox object model show how to get from one object type to another. To take one example: if you are only interested in doing a database search without worrying about what Cardbox is currently displaying, use the `Databases` property to get a Database object for the database you're interested in, or use `Windows.OpenFile` to open the database if it isn't already open. On the other hand, if your program is planning to drive Cardbox in a way that is visible to the user, then use the `Windows` property to locate a named Window object or `ActiveWindow` to get a Window object for the active window.

Improving speed

Be economical about method and property calls as far as you can, because they take more time when they come from an external program. Suppose that you want to set the values of three fields in the current record. You might be tempted to do this:

```
cbx.ActiveWindow.Fields("FIELD1")=1
cbx.ActiveWindow.Fields("FIELD2")="xxx"
cbx.ActiveWindow.Fields("FIELD3")=123
```

but it is enormously inefficient. The `ActiveWindow` method will be called three times in a row, as will the `Fields` method, and the indexing operation that extracts a Field object from the Fields object, and the assignment to the implicit `Text` property of the Field object. Compare this:

```
Set flds=cbx.ActiveWindow.Fields
flds("FIELD1")=1
flds("FIELD2")="xxx"
flds("FIELD3")=123
```

If you really need to improve performance, you can set all the field values with just one command. See Help Point 753.

Now the `ActiveWindow` and `Fields` methods are called once only. If you are setting many field values and processing many records, this will save a lot of time.

Translating from Cardbox macros

When you record a macro in Cardbox there are practically no object references, because Cardbox knows that the macro will be run in Cardbox's own macro environment, which inserts object references for you: thus it will happily record `ClearTagged` even though the full command, with object references, is really `ActiveWindow.Database.ClearTagged`. It's often useful to start programming by recording an operation in Cardbox and then translating it into your own language, but if you do this then use the Index of Methods and Properties on page 132 to make sure that you insert the right object references: your language won't insert them for you.

You will also see that Cardbox uses constants such as `cbxWriteFormatCSV` to control the way that various commands work. Your programming language may well import values for these constants from the built-in Cardbox definitions, but if it doesn't then you will have to replace the constants by numbers or define their values for yourself: see Help Point 762.

Visual Basic

These notes relate to Visual Basic 6.0, but other versions may be similar.

When you open a Visual Basic project that you intend to use with Cardbox, enter the command Project > References and scroll through the list of available references until you find “Cardbox”. Turn this option on. (If you also have a reference to “Cardbox Type Library”, ignore it because it refers to Cardbox 2.0).

You will then be able to declare variables with any of the Cardbox object types:

```
Dim cbx As Cardbox.Application
Set cbx=GetObject(,”Cardbox.Application”)
Dim win As Cardbox.Window
Set win=cbx.Windows(“LETTERS”)
```

Intellisense doesn’t work with the Print method, presumably because Print is also a command in Visual Basic. This Print method will still work despite this.

Declaring variables like this has several advantages:

- Visual Basic can display Intellisense™ prompts and allow auto-completion while you are typing your program.
- You can use Cardbox’s built-in constants in your program.
- Visual Basic can check, before you run your program, that you aren’t trying to store the wrong type of object into the wrong type of variable or using the wrong method or property for the object type. (VBScript can’t do this because VBScript variables don’t have fixed types).

VBA in Microsoft Office

These notes relate to Word 2002, but other Office applications are similar.

VBA (“Visual Basic for Applications”) is the macro language used by Microsoft Office. It is a close cousin of Visual Basic and can be used to drive Cardbox in just the same way.

When designing a VBA macro that will be used with Cardbox, enter the command Tools > References in the Microsoft Visual Basic window, scroll through the list of available references until you find “Cardbox”, and turn this option on. You will then be able to declare variables with any of the Cardbox object types, while at the same time having access to the entire Office object model. Here is a simple example:

```
Dim cbx As Cardbox.Application
Set cbx=GetObject(,”Cardbox.Application”)
Selection.TypeText cbx.ActiveWindow.ActiveRecord.Fields(“NA”)
```

When you play this macro in Microsoft Word, it does the following:

1. It gets a reference to the currently running copy of Cardbox.
2. It uses Cardbox’s `ActiveWindow`, `ActiveRecord`, and `Fields` methods to get the contents of the field called NA in the current record in the currently active window in Cardbox.
3. It types the text of this field into Word, at the current cursor position.

This straddling of the object model of several applications is what makes VBA macros so useful when used with Cardbox.

- A bug in Office means that the `Left` function (used to extract substrings from strings) does not work in a macro that incorporates the Cardbox object model, presumably because Cardbox defines a property called `Left` and VBA gets confused by this. The way round the bug is to say `VBA.Left` instead of `Left`. `VBA.Left` always works perfectly in a Word macro.

VBScript in Windows

The Windows facility that makes this possible is called the Windows Scripting Host, or WSH for short.

Apart from using VBScript in Cardbox, you can also use it in Windows itself. If you write a program (a “script”) in VBScript and give it the filetype `.vbs`, then double-clicking on it will execute the script. Some people like using VBScript in this way because running a script is then an action that is controllable by Windows itself – so that you can, for instance, schedule a script to be run daily at a specific time.

In this context VBScript doesn’t have any of the privileged access to commonly used method and property names that it does when it’s used inside Cardbox, so you will have to work your way through the Cardbox object model just as you do with Visual Basic.

The main thing to be careful of is that Cardbox’s built-in constants are not made available to you automatically. So if you say something like

```
win.Print cbxPrintMainRecord
```

you’ll get an error. (If you are using Option Explicit, VBScript will complain that a variable hasn’t been declared; if you aren’t, VBScript will use 0 as a default value for `cbxPrintMainRecord`, but this is not a valid option for the `Print` method and Cardbox will complain about an invalid argument).

The cure for this problem is to import Cardbox’s constant definitions into the start of your script: you can find them through Help Point 762.

VBScript in web pages

VBScript was originally designed as a client-side scripting language for web pages, and so you can certainly put VBScript code in your web pages that accesses or drives Cardbox. In practice it is of limited use. First of all, it depends on using one manufacturer’s web browser and therefore can’t be used on the general Internet; but more importantly, Internet Explorer’s security settings make it quite difficult to run this kind of script without running into an immense number of warning messages. In any case, if you are running a Windows program to access Cardbox content, you might as well use Cardbox itself and not a painstakingly crafted web page: the Client Edition of Cardbox is, after all, freely downloadable.

VBScript in Active Server Pages

Active Server Pages (ASPs) are web pages that are processed by your web server in some way before they are sent to the requesting web browser. Using VBScript to access Cardbox from within an ASP has some advantages.

1. You have access to all the power of Cardbox and can make it available to web users even if they aren't running Windows. You can do anything: from a simple data lookup to help fill in some of the data in your page, to a sophisticated HTML form interface that lets people search your database using their web browsers.
2. Because nothing happens at the browser end, there are no security or compatibility issues to worry about. By the time the ASP page has arrived at the browser, it is ordinary standard HTML that any browser will understand.

Setting up ASPs and linking them to Cardbox databases requires a sound knowledge of the workings of Active Server Pages and of your web server, and we're not going to go into more detail on this subject here. Help Point 765 suggests some sources of further information.

Other programming languages

Visual Basic and its cousins are not the only languages that can handle objects: in fact, most modern programming languages can do this in one way or another. We are not going to go through the details of interfacing them all to Cardbox because the principles are essentially the same whatever the language, and the details often vary widely between vendors: you should check the documentation for details of COM (Common Object Model) support.

Cardbox objects – the GetObject and CreateObject functions that we have described are specific to the Visual Basic family, and you will have to consult your own language's documentation to find out how to connect to existing Cardbox objects or create new ones. Help Point 764 has some suggestions.

Type library support – if your language lets you import type libraries (or "object libraries"), then it will be able to import the type library for Cardbox. Do this if you can, because it will make your programming easier (for example, it may mean that your development environment can identify the Cardbox object types and know which methods and properties are valid for each).

Constant definitions – if your language can't get valid values for the Cardbox constants such as `cbxPrintMainRecord`, you may need to add the necessary definitions by hand: see Help Point 762.

Part Seven
Examples and strategies

Examples of Cardbox macros, with explanations of how they work.

Index of examples

Some quick actions

Telephone dialling

See Help Point 562.

If you have the right autodialling software and hardware installed, the following macro will dial the number stored in the field called TEL. A more sophisticated version could process or reformat the number before it is dialled: for example, by inserting or removing a dialling code.

```
Dial Fields("TEL")
```

Opening web pages

You can use the same technique to send emails without needing to enter the mailto: prefix.

If you prefix a web address with `http://` then Cardbox will underline the address and will open the page as soon as you click on it. If you don't like to see these prefixes on the screen, you can design a macro like the following one, which opens the web page whose address is stored in a field called WEB. You could design a button in the format, or assign a keystroke to make it easy to run this macro.

```
Launch "http://" & Fields("WEB")
```

You can extend this quite easily: suppose that you can have several web addresses in a field and want to be able to open all of them together. Here is a macro that does this:

```
webs=Split(Fields("WEB")," ")
For i=LBound(webs) To UBound(webs)
 Launch http:// & webs(i)
 Next
```

Opening files on your computer

This technique is specially useful for MP3 music files.

Sometimes you'll want your Cardbox databases to refer to other files stored on your computer. For example: we store our invoices as Word files, and our sales ledger database in Cardbox shows an invoice number for each record. To open the invoice for a given record, we can use a macro like this:

```
Launch "C:\Office\Invoices\" & Fields("IN") & ".doc"
```

Windows provides its own mechanism, OLE ("Object Linking and Embedding") to support links to files, but it is cumbersome and bureaucratic, so this "poor man's OLE" is often the quickest and most effective solution. We can easily elaborate the macro to handle more complex cases: for instance, if invoices of more than a certain age are archived and stored in another location.

Field manipulation while editing

To copy one field to another within the current record, use a macro like this:

```
Fields("DEST")=Fields("SRC")
```

To copy several fields at once, use something like this:

```
Fields("DEST1,DEST2")=Fields("SRC1,SRC2")
```

To copy a field from another record (whose Record object you have previously stored in a variable called `rec`), use a macro like this:

```
Fields("XXX")=rec.Fields("XXX")
```

If you want to add (append) the text of one field to the end of another, see page 71.

For an example of field arithmetic, see page 23.

If you want to total a list of numbers in one field (say, LIST) and store the result in another field (say, TOT), here is one way of doing it:

```
nums=Split(Fields("LIST")," ")
total=0
For i=LBound(nums) To UBound(nums)
 total = total + nums(i)
 Next
Fields("TOT")=total
```

Intelligent record duplication

Cardbox lets you create a blank record with Edit > New Record and it lets you create an exact duplicate of the record you're looking at with Edit > Duplicate Record, but it doesn't do anything in between: it can't duplicate some fields but not others. Here is a macro that will do this for you.

```
Set recOld=ActiveRecord
NewRecord
Fields("TITLE,PUB,AUTHOR")=recOld.Fields("TITLE,PUB,AUTHOR")
```

Automatic calculations

You may want some actions and calculations to happen automatically when a record is saved (as if you were creating your own validators). Cardbox doesn't allow you to override its built-in commands like this, but you can get a similar effect by programming a macro to be activated with the Ctrl + S keystroke and then training your users to use the keystroke rather than the File > Save command.

Building macros that use objects

Changing an existing macro to use objects

One Cardbox user has the following problem: he has two databases, open in windows called PROC and EMAIL. They both have the same number of records selected, and he has to transfer the contents of one particular field (called AB) from each record in EMAIL to the corresponding record in PROC. Here was his first solution, obtained by simply recording the actions he was taking manually and adding a `For / Next` loop to it:

```
For pos=1 to ActiveWindow.RecordCount
GoToRecord pos
EditRecord
GoToField "AB"
Command cmdSelectAll
Command cmdCopy
DontSaveRecord
Activate "PROC"
EditRecord
GoToField "AB"
Command cmdSelectAll
Command cmdDelete
Command cmdPaste
SaveRecord
NextRecord
Activate "EMAIL"
Next
```

The big trouble with this macro is that it is slow. If you think through what it's doing then it's not difficult to see why. It has to step through every record in two databases, and for each record it has to switch from one window to another and back again. It also has to edit one record just to get the contents of one of its fields.

Using Records for maximum speed

If you forget about the recorded commands and look at what the core of this macro is actually doing, you'll come up with something like this:

1. Look at the *n*th record in the EMAIL database, and extract the contents of field AB.
2. Edit the *n*th record in the PROC database, store the extracted text into *its* field AB, and save it.

So let's make a subroutine that does just that:

```
Sub CopyABField(recTo,recFrom)
 recTo.Edit
 recTo.Fields("AB")=recFrom.Fields("AB")
 recTo.Save
 End Sub
```

We need to get those Record objects from somewhere, of course. We start by getting the Records object that represents the current selection in EMAIL, and the same for PROC:

```
Set recsEMAIL=Windows("EMAIL").Records
Set recsPROC=Windows("PROC").Records
```

Then we store the record count in a variable, for convenience, and take the opportunity to check that the records counts of both databases are the same:

This check wasn't in the recorded macro but it's a good safety feature.

```
nRecs=recsEMAIL.Count
If recsPROC.Count<>nRecs Then
  Halt "The record counts don't match"
  End If
```

Finally we loop through the records and process each one:

```
For pos=1 To nRecs
 CopyABField recsPROC(pos),recsEMAIL(pos)
 Next
```

And that's it. The original 17-line macro has been reduced to 14 lines, but more importantly it's far faster because it all happens behind the scenes. It's also easier to understand and modify: it took several readings of the original macro for us to be certain that it was copying from EMAIL to PROC and not the other way round; and if we ever wanted to copy two fields instead of one, that would require an extra nine lines in the original macro but only one line in the revised one that uses objects.

Like all programming, this macro represents an investment. If you are only doing something once, then the original inefficient version may actually be your best bet: because you know it works and you can always go out to lunch while it's processing your large database. If, on the other hand, this is a process that you will be going through once a week, then it is well worth investing the time in going all the way to the Records version and getting good performance and a reliable macro that will be easy to modify or expand later.

Using a second database as a lookup table

If you have a whole batch of records to process at once, see page 106 for a faster way of doing lookup tables.

Page 61 described how a user with multiple shops wanted to be able to select a shop from a drop-down list and fill in a number of fields based on his selection.

Let's assume that the field used for the selection is called SCODE and that the fields to be filled in are called SNAME, SADDR, and STEL. The separate database that lists the shops is already open in a window called "Shop Addresses", and for simplicity we'll assume that the field names are the same in both databases, although they don't have to be.

```
shopcode=Fields("SCODE")
Set recs=Windows("Shop Addresses").Database.AllRecords
Set recs=recs.Select("SCODE",shopcode)
If recs.Count<>1 Then Halt "Invalid shop code!"
Fields("SNAME,SADDR,STEL")=recs(1).Fields("SNAME,SADDR,STEL")
```

1. The macro extracts the shop code from the current record into `shopcode`.
2. It stores all the records in the Shop Addresses database in a variable called `recs` and then reduces `recs` to just the records whose SCODE field matches `shopcode`.
3. There should only be one record like this: if there are none, or if there is more than one, then there is something wrong with either the shop code or the Shop Addresses database, so the macro halts and reports an error.
4. The macro copies fields from the first (and only) record that it has selected into the record that the user is currently editing.

If the SCODE field is done as a drop-down list, it will be impossible for the user to enter an invalid shop code.

Tiny lookup tables

We've used just one field here, to make the example shorter.

Here is a version of the same macro for when there are only a few cases to be covered. It doesn't need a second database, but the macro will need to be modified every time a new shop is opened.

```
Select Case Fields("SCODE")
 Case "BUD" : Fields("SNAME")="Budleigh Salterton"
 Case "CHIT" : Fields("SNAME")="Chitterne"
 Case "HOL" : Fields("SNAME")="Holcombe"
 Case Else : Halt "Invalid shop code!"
 End Select
```

Using a second database as a thesaurus

This macro was used in a database of accidents. All the contributory factors to each accident needed to be listed in a singe field, and codes were assigned to every possible factor so that it would be easy to analyse the date. There were too many codes for the users to remember them perfectly, and check boxes were not an option – again, because of the number of codes: the check boxes would have taken up too much space on the screen.

The solution was to maintain a second "thesaurus" database that listed what codes were allowed and gave enough explanatory information for the user to know which codes to choose. The basic instructions, as given to the users, were:

1. Go to the field that you want to enter codes in, and press Ctrl + L.
2. Cardbox will list all possible codes for the field you are using, and will then pause. Tag the ones you want, then press F5 to continue.
3. Your chosen codes will be typed into the field.

Tools > Keyboard was used to program Ctrl + L to play this macro.

Here is the macro that did all this. We assume that the thesaurus database is open in a window called "Thesaurus". The thesaurus database has three main fields: FIELD identifies which field in the main database was being referred to, CODE gives the code to be typed into that field, and EXPLANATION gives the user an explanation of what exactly this code means and when to use it.

```
windowname=ActiveWindow.Name
fieldname=ActiveFieldName
Activate "Thesaurus"
SelectionLevel=0
ClearTagged
Select "FIELD",fieldname
If RecordCount=0 Then
  Activate windowname
  Halt "Field " & fieldname & " has no thesaurus entries."
  End If
Pause "Tag the terms you want to use, then press F5"
Set recs=Database.TaggedRecords
Activate windowname
For Each rec In recs
 TypeText rec.Fields("CODE") & " "
 Next
```

1. The macro notes the name of the active window and the active field within that window.

2. The macro switches to the Thesaurus window. The next few commands will be sent to that window.
3. The macro clears all tags, resets the selection to Level 0 and selects just the records that relate to the field the user was editing when he started the macro.
4. The macro pauses, with a message.
5. The user now goes through the selected records and tags the ones he wants. He could also make selections among the records, if that made finding the right ones easier; as long as all desired records end up being tagged.
6. The user continues the macro.
7. The macro stores a Records object containing the tagged records into the variable `recs`.
8. The macro switches back to the original database (its name was noted at the beginning of the macro).
9. The macro looks through each of the tagged records in `recs`. For each record, it types the contents of the CODE field into Cardbox (just as if it was typed directly by the user) and types a space after each one so that the codes are kept separate.

If you want to adapt this macro for your own use, and you only have one field that should have thesaurus codes attached, then you don't need a field called FIELD in the thesaurus database and you can remove all references to `fieldname` from the macro.

Advanced techniques

Using temporary fields

It can sometimes be useful to create a temporary field that doesn't add anything to the information that is already in your records but simply formats it in a different way. Here's an example:

Suppose that you have a list of names and addresses and want to sort it into alphabetical order using the same rules as British telephone directories. These say, among other things, that all forms of the "Mac" prefix (Mac, Mc, M') are sorted as if they were spelt "Mac". Similarly, "St" is expanded into "Saint".

To deal with this, create a separate field, which we'll call NFS ("Name For Sorting"). The idea of NFS is that it should contain a version of the person's name that has been transformed so that if you get Cardbox to sort your records using the NFS field, they will come out in the exact sequence you want. For instance, if the name is "M'Turk" then NFS will have to contain "MacTurk"; if it is "St John" then NFS must contain “Saint John”. (NFS will never be printed out: its only purpose is to make sorting produce the result that you want).

Obviously you don’t want to have to fill in all the NFS fields by hand: this is tedious and error-prone. So create a macro that looks at the person’s real name and transforms it into its NFS form. Most of the time this will simply mean copying the name into NFS, unchanged; but the macro will check for and transform the “Mac” and “Saint” prefixes, and it can do any other changes you want – you can even program it to turn “8” into “Eight” if that’s what the users of the printed list will expect and if you have the patience to do the programming.

Once the macro has been written, printing your address list in the right order is a simple three-step process:

1. Run the macro to fill in the NFS field for every record.
2. Sort the records into order, using the NFS field.
3. Print out the records in an appropriate format.

The “Needs Change” flag

Many of the techniques we’ve shown you involve calculating fields from other fields: either arithmetically, or by looking things up in another database, or by transforming the data programmatically. These things take time, and if you have a huge database then it will be a great waste of time to repeat these calculations when only a handful of records may have changed since the last time you processed them.

There is a simple general way round this problem:

1. Create an indexed field called (say) CHANGED, and set up a Default Value validator that writes “Y” into this field whenever the record is saved.
2. In the macro that performs calculations on all your records, select just those records that have “Y” in CHANGED, and process them, then empty the CHANGED field just before saving the record, like this:

The Save method of the Record object doesn’t activate validators, so your Default Value validator won’t put “Y” into CHANGED when the macro saves the record.

```
For rec In Database.AllRecords.Select("CHANGED","Y")
 rec.Edit
 ‘ Perform other processing on the record ‘rec’
 rec.Fields("CHANGED")=""
 rec.Save
 Next
```

Making lookup faster

See the example with shops on page 102.

One Cardbox user regularly imports trade statistics from a corporate system and uses Cardbox to process them. This processing includes looking at a country code and filling in country names and other details on the basis of that code. We've already shown you how to do this sort of thing, but the snag here is that there could be a million records in the database; and even though Cardbox is fast, a million database searches will take up a lot of time.

The key to overcoming this problem is that although there may be a million records there certainly aren't a million countries. Assuming that the country code is indexed, here are two ways of making things faster.

Using ListIndex

Instead of ListIndex and Split, you could use ListIndexToFile to write the codes to a file, and then read the codes from that file and process them one at a time. This would make more sense if you had a very large number of codes.

Use the ListIndex method to extract a list of all the country codes that are indexed in this database. Use Split to split the list into an array in VBScript and then, for each country code:

1. Select the records for that country code.
2. Look up the data for that country code in the reference database and store them in VBScript variables.
3. Loop through your selected records and fill in the relevant fields using the data that you have stored in your variables.

This is faster because the search in the reference database is done only once for each country code and not once for each record in the big database.

Using Exclude

Here is the outline of an alternative approach. We assume that the country code is stored in a field called CC, and we've invented a few other field names so that you can see how the whole thing would work.

```
Set recs=Records
Set recsRef=Windows("Country Data").Database.AllRecords
While recs.Count>0
 ccode=recs(1).Fields("CC")
 Set fldsFound=recsRef.Select("CC",ccode)(1).Fields
 cname=fldsFound("CN") : cdata=fldsFound("CD")
 Set recsToProcess=recs.Select("CC",ccode)
 If recsToProcess.Count=0 Then Halt
 For Each rec In recsToProcess
  rec.Edit : flds=rec.Fields
  flds("CN")=cname : flds("CD")=cdata
  rec.Save
  Next
 Set recs=recs.Exclude("CC",ccode)
 Wend
```

1. The variable `recs` contains the records that haven't been processed so far. In this example the macro processes all records in the current selection. You could process all records in the database using `Database.AllRecords`, or you could follow the "Needs Change" example on page 105 to reduce the number of records you need to process.
2. The macro stores a reference to the lookup table in `recsRef` to reduce the number of times that the table has to be retrieved.
3. The macro looks at the first record in `recs` and selects matching records from the lookup table. There should be exactly one record: if there are more then it will use the first record; if there are none then the macro will halt with an error.
4. The macro extracts the values it needs from the lookup table.
5. Now the macro selects all records in `recs` that have the same CC value as the first one in the selection, and processes all those records to insert the data into them.
6. Finally, the macro removes from `recs` all the records with this CC value: in other words, all the ones it has just processed. The whole procedure repeats itself as long as there are records remaining in `recs`.

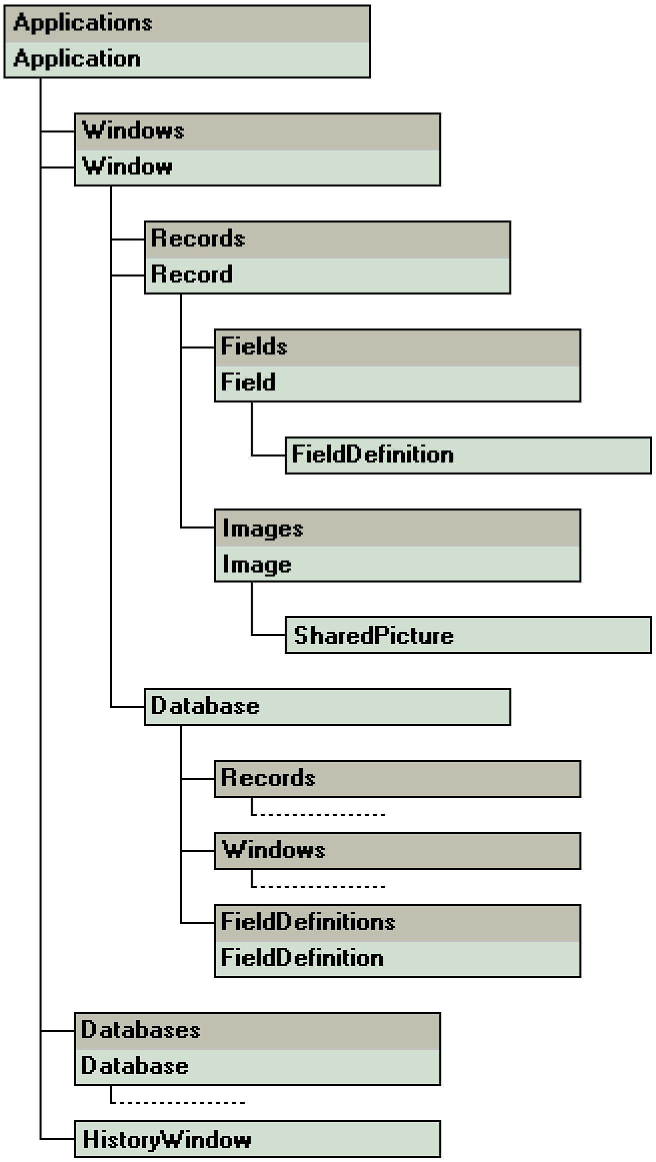
As always with macros that use objects, you'll see virtually nothing happening on the screen while this macro is running.

Part Eight

The Cardbox object model

A list of all Cardbox objects, with a summary of their interrelations, methods, and properties.

Applications: p.113
Application: p.113
Databases: p.128
Database: p.128
Fields: p.124
Field: p.125
FieldDefinitions: p.130
FieldDefinition: p.130
HistoryWindow: p.114
Images: p.126
Image: p.127
Records: p.121
Record: p.122
SharedPicture: p.127
Windows: p.115
Window: p.116



The lists given here tell you what the methods and properties of each object type are, but they don't give any details of the way that they are to be used. To get more information, you have two choices:

- Look up the relevant Help Point and follow the links to a full specification in the help file.
- Or, for methods that correspond to commands, press the Record button, perform the command yourself, and then look at what the macro recorder has recorded for you.

Help Point 737. Built-in methods and properties

Some methods and properties are built in to the macro system. You don't need an explicit object reference to use them. They are accessible only from macros running within Cardbox: you can't use them from an external program.

Properties

`SafetyLevel`	The safety level of the current macro. In some restricted cases you can modify this property as well as retrieving it: see "Safety Levels" on page 9.
`StatusText`	The text shown in the Cardbox status bar.
`StatusPosition`	The length of the blue progress bar in the Cardbox status bar: `StatusPosition=0` indicates zero length and `StatusPosition=StatusRange` indicates that the entire progress bar is blue.
`StatusRange`	If this is zero then the progress bar in the Cardbox status bar contains a moving block; if it is non-zero then it contains a blue bar whose length is controlled by `StatusPosition`.
`StatusAutomatic`	If you set this to True, `StatusRange` reflects the number of records in the current selection and `StatusPosition` reflects the current record position.
`CommandLine`	When you program a keystroke, toolbar button, etc to play a macro, you can include a "command line" along with the macro name. This property reflects the command line that was included and your macro can examine it to decide what action to take.

`Scrap(`***n***`)`	`Scrap(1)` to `Scrap(100)` are 100 temporary storage locations that you can use to store data. The values stored here will persist until this copy of Cardbox is closed, so you can use them to communicate information between one macro and another.

Methods

`Halt`	Terminates the macro, optionally displaying a message.
`Pause`	Pauses the macro, optionally displaying a message, and waits for the user to restart it.
`Play`, `PlayText`	Plays a macro and waits for it to complete. Play identifies the macro by name and PlayText explicitly gives the text of the macro to be played.
`Launch`	Opens an external program or document.
`Run`	Opens an external program and waits for the program to terminate before continuing the macro.
`Sleep`	Does nothing for a specified length of time. This can be useful when driving external programs that require a pause between commands (eg. some types of fax software).
`WriteLog`	Writes a line to Cardbox's debug log. This can be useful when you are developing or testing a macro and trying to see exactly what happens when.
`Dial`	Dials a given telephone number. This depends on suitable hardware being installed on your computer along with an appropriate telephone dialler program that supports the Simple TAPI interface. (Such a program is built in to Windows XP).
`Connection`	Creates and returns a Connection object (see page 131).
`GetMailExchangers`	Given the domain name part of an email address (the part after the @ sign) returns an array containing a list of domain names that are ready to receive email for that address. Requires Windows XP.
`DateFromCardbox`	Converts a date in one of the textual date formats understood by Cardbox to VBScript's built-in Date type. See "Regional Settings" on page 35.
`DateToCardbox`	Converts a value of VBScript's built-in Date type to one of the textual date formats understood by Cardbox.

`NumberFromCardbox`	Converts a value from string to number type. See “Regional Settings” on page 34.
`NumberToCardbox`	Converts a value from number to string type. See page 34.

Help Point 720.

The Applications object

The Applications object is a collection of Application objects representing all the copies of Cardbox that are currently running on your computer.

This is a very specialised object and we cannot imagine you ever needing to use it from a macro. We have provided it in case you want to write a separate program that looks at the running copies of Cardbox and chooses which one to connect to.

Help Point 721 gives full details of how to get and use an Application object and includes links to articles on each property and method.
See also page 62.

The Application object

The Application object represents a copy of Cardbox that is currently running. Its methods and properties let you do things such as resizing the Cardbox window, and it is also the starting point for getting objects that refer to database windows and the databases and records themselves.

If you are running a macro within Cardbox then you won’t often need the Application object, because most of the properties it provides are accessible to macros directly; but if you are driving Cardbox from an external program then the Application object is important because it is the first object that you get hold of when navigating through the Cardbox object hierarchy.

Getting an Application object

In macros, `Application` will give you the application object directly.

With external programs, you will need to use `CreateObject` or `GetObject` to get hold of an Application object. The exact syntax depends on the programming language and on whether you are trying to refer to a copy of Cardbox that is already running or trying to start a new copy of Cardbox: see page 89.

All Cardbox objects have an `Application` property that gives you the Application object that they belong to: this is present for completeness and you wouldn’t normally need to use it.

Properties

The following properties are read/write: you can both set them and retrieve them.

`Left`, `Top`, `Width`, `Height`	The position and size of the main Cardbox window. You can't change these values when the window is maximised.
`WindowState`	The window's state (maximised, minimised, or normal).
`Caption`	The window's caption.
`Visible`	This is True if the Cardbox window is visible (which it normally is) or False if it is invisible.
`ClipboardText`	The text currently in the Windows Clipboard.

The following properties are read-only: you cannot change them.

`Context`	The current context of the Cardbox user interface: editing records, dialog box open, etc.
`Name`, `Build`, `BuildNumber`	Information about this version of the Cardbox program.
`MachineName`	Identification of the computer that this copy of Cardbox is running on.
`Workspace`	The filename of the workspace file.

Object properties

The following properties give access to other Cardbox objects. When you use them in a macro, there is no need to prefix them with `Application` because Cardbox will automatically understand that they refer to the Application object.

`Windows`	A Windows object that is a collection of all the database windows that are currently open in this copy of Cardbox.
`ActiveWindow`	A Window object that refers to the currently active database window within Cardbox.
`Databases`	A Databases object that is a collection of all the databases that are currently open in this copy of Cardbox. There is no ActiveDatabase property. If you need the Database object for the currently active database, use `ActiveWindow` to get a Window object for the active window and then use the window's `Database` property to get the Database object.
`HistoryWindow`	A special HistoryWindow object that lets you control the position and visibility of the History window. For details of its methods and properties, see Help Point 722.

Methods

`Activate`	Activates the Cardbox window. You won't normally need it in a macro because when you are running a macro, Cardbox is already the active application; but it can be useful when you are driving Cardbox from an external program.
`Play`	Plays a named macro.
`PlayText`	Plays specified text as a macro.
`SaveRegistryOptions`	Saves Cardbox's registry options to a file, like Tools > Management > Save Cardbox Registry Settings.
`DateFromCardbox`, `DateToCardbox`	See page 35. These Cardbox functions are provided as methods of the Application object so that external programs can access them

The Windows object

Help Point 723 gives full details of how to get and use a Windows object and includes links to articles on each property and method.
See also page 63.

The Windows object is a collection of Window objects representing database windows within Cardbox. It can represent:

- All the windows open within Cardbox, if the Windows object was obtained from the `Windows` property of an Application object.
- All the windows open on a particular database, if the Windows object was obtained from the `Windows` property of a Database object.

The Windows object is mostly used to collect together Window objects, but it has a few methods of its own, of which the most important is `OpenFile`, which opens a Cardbox database in a new window.

Properties

The Windows object has the standard properties shared by all collections:

`Count`	The number of Window objects in this collection.
`Item(`*n*`)`	The *n*th Window object in this collection.
`Item("`*name*`")`	The Window object for the window whose caption is *name*.

Methods

`Cascade`	This cascades the database windows within a Cardbox window, just like Window > Cascade.
`Tile`	This tiles the database windows within a Cardbox window, just like Window > Tile.
`OpenFile`	This opens a Cardbox database in a new window. It returns a Window object that refers to the new window.

The Window object

Help Point 724 gives full details of how to get and use a Window object and includes links to articles on each property and method.
See also page 63.

The Window object represents a database window within Cardbox.

- The Windows object lets you get a Window object for any database window at all.
- The `ActiveWindow` property of the Application object represents the currently active database window.

The Window object has an enormous number of methods and properties, because almost any menu command that you enter into Cardbox is effectively an action applied to the currently active window. For example, going to the 10th record in the current selection of the currently active window is done by the command:

```
Application.ActiveWindow.GoToRecord 10
```

Because so many commands refer to the active window (especially when you record them in macros), the macro system lets you abbreviate this sort of command to:

```
GoToRecord 10
```

Exception: you have to say ActiveWindow.Left and ActiveWindow.Select, because Left and Select are keywords built into Visual Basic.

In the macro system, this applies to all properties and methods of the Window object: if you leave out any object reference, the active window is assumed.

When you are programming from another language you don't have access to this short cut, and you will need to get hold of a Window object explicitly if you want to use its methods and properties.

Properties

`Left`, `Top`, `Width`, `Height`	The position and size of the database window. You can't change these values when the window is maximised.
`WindowState`	The window's state (maximised, minimised, or normal).
`Caption`	The window's caption.

`Visible`	This is True if the window is visible (which it normally is) or False if it is invisible.
`Index`	This window's position in Cardbox's Window menu.
`ZOrder`	The position of this database window relative to other database windows within Cardbox. The currently active window has `ZOrder=1`; the window underneath it (possibly completely obscured by it) has `ZOrder=2`; and so on. You can't change the value of `ZOrder` but you can move a window to the top by using the `Activate` method to activate the window.
`Name`	The name (caption) of the window.
`Caption`	An exact synonym for `Name`.
`WindowNumber`	If several windows have the same name, Cardbox gives them suffixes: #1, #2, and so on. `WindowNumber` gives the number in the suffix.
`Format`	The name of the format this window is using. The native format is identified by a zero-length string "". To change to a different format, set this property to the name of the new format: for example, `Format="REPORT"` will change this window to use the format called REPORT (if such a format exists).
`ExtraText`	Sets the display mode for extra text: main record only, extra text only, or main record and extra text together.
`HighlightMatches`	Checks or sets the mode for highlighting matching words after a search.
`Sequence`	The sequence into which records have been sorted. You can sort records by setting the `Sequence` property (see Help Point 752) or you can use the `SetSequence` method, which has some additional options.
`SelectionLevel`	The number of active levels of selection (this is the Level value reported in the status bar at the bottom of the Cardbox window). Setting this property to less than its current value makes Cardbox undo levels of selection until it gets to the level you have specified. You can't explicitly set it to more than its current value.
	Reducing `SelectionLevel` by 1 is such a convenient operation (it is equivalent to Search > Undo) that a method exists for doing it: `UndoOneSelectionLevel`.

`StepBrowseMode`	This is `True` if step browse mode (from the Search > Browse menu) is active and False if it isn't. You can cancel step browse mode by setting this property to `False`.
`Editing`	This is `True` if this window is currently editing a record or `False` if it isn't. You can use this so that your macro can take different actions in the two cases. Often you may find yourself with a macro that can only work when Cardbox is editing (or conversely, when Cardbox is not editing). In that case it is good practice to check the editing mode and terminate at once if it is wrong: you can use the `Editing` property for this, or the `CheckEditing` method can do it for you in a single step.
`RecordPosition`	The position in the current selection. To move to a different record you can use the `FirstRecord`, `LastRecord` `NextRecord` and `GoToRecord` methods, or you can simply give a new value to `RecordPosition`.
`RecordCount`	The number of records in the current selection. This gives the same value as `Records.Count` but without the extra work of creating a temporary Records object.
`ActiveFieldName`	When you are editing a record, this is the name of the field you are currently in. This gives the same value as `ActiveField.Name` but without the extra work of creating a temporary Field object.

Object properties

The following properties give access to other Cardbox objects.

`Database`	A Database object that represents the database that is open in this window.
`Records`	A Records object that is a collection of all the records in the current selection.
`ActiveRecord`	A Record object that represents the current record in the current selection.
`ActiveField`	When you are editing a record, this is a Field object representing the field that you are in.

Methods

`Activate`	Activates this window.
`Close`	Closes this window.
`SetSequence`	Sorts the current selection into a given sequence.
`UndoOneLevelOfSelection`	Undoes one or more levels of selection.
`CheckEditing`	Checks that the current editing mode (`True` or `False`) matches the one you specify, and terminates the macro at once if it doesn't.
`FirstRecord`, `NextRecord`, `LastRecord`, `GoToRecord`	These methods move to different record positions in the current selection.
`Print`	Prints a record or records, like File > Print.
`Records.WriteToFile`	Export records to a file, like File > Export. Note that `WriteToFile` is actually a method of the Records object, which is why the reference to the `Records` property is needed.
`ReadFromFile`	Imports records from a file, like File > Import > From File.
`ReadFromCardbox`	Imports records from another window, like File > Import > From Cardbox.
`DeleteRecord`	Deletes the current record, like Edit > Delete Record.
`UndeleteRecord`	Undoes the most recent deletion, like Edit > Undo Deletion.
`RemoveDeletedRecords`	Removes deleted records from the current selection, like View > Remove Deleted Records.
`AddRecord`	Adds a new record and starts editing it, like Edit > New Record.
`EditRecord`	Starts editing the current record, like Edit > Edit Record.
`DuplicateRecord`	Duplicates the current record, like Edit > Duplicate Record.
`SaveRecord`	Saves the record you were editing, like File > Save Record.

`DoNotSaveRecord`	Stops editing but does not save the record, like File > Quit Without Saving.
`GoToField`	When editing, moves to a different field.
`TypeText`	When editing, types text into the field.
`TypeTextFromFile`	When editing, reads text from a file and types it into the field.
`Command`	When editing, performs various editing keystrokes.
`Find`, `FindNext`, `ReplaceAll`, `ReplaceHere`	When editing, implements find-and-replace commands.
`SetOption`	When editing, selects a particular option in a check box, radio button, or drop-down list field.

Search and selection methods

These following methods are the equivalent of commands in the Search menu: `Select, SelectKept, SelectTagged, SelectData, SelectRecordNumber, SelectFromWindow, SelectRelational, SelectAlso, SelectAlsoKept, SelectAlsoTagged, SelectAlsoData, SelectAlsoRecordNumber, SelectAlsoFromWindow, SelectAlsoRelational, Exclude, ExcludeKept, ExcludeTagged, ExcludeData, ExcludeRecordNumber, ExcludeFromWindow, ExcludeRelational, ExcludeAlso, ExcludeAlsoKept, ExcludeAlsoTagged, ExcludeAlsoData, ExcludeAlsoRecordNumber, ExcludeAlsoFromWindow, ExcludeAlsoRelational, Include, IncludeKept, IncludeTagged, IncludeFromWindow, IncludeRelational, StepBrowse, StepBrowseKept, StepBrowseTagged, StepBrowseData, StepBrowseRecordNumber, StepBrowseFromWindow, StepBrowseRelational.`

You can also build up a selection within a Records object and then combine it into the current selection using:

`SelectFromRecords, SelectAlsoFromRecords, ExcludeFromRecords, ExcludeAlsoFromRecords, IncludeFromRecords, StepBrowseFromRecords`

or you can use `SetFromRecords` to replace the window’s current selection with the records you have placed into your Records object.

Help Point 725 gives full details of how to get and use a Records object and includes links to articles on each property and method.
See also page 64

The Records object

The Records object is a collection of Record objects representing records in a database. It can represent:

- All the records in a database.
- The tagged records in a database.
- The records in the current selection in a window.

It is also possible to build Records objects that have no direct relation to anything that is visible on the Cardbox screen, and this is an important technique when you are writing macros or programs to do searches independently of the user's view of Cardbox: such programming avoids distracting flickering on the screen and can also be much faster.

Properties

The Records object has the standard properties shared by all collections:

`Count`	The number of records in this collection.
`Item(`*n*`)`	The *n*th record in this collection. This is a Record object.

Object properties

The following property gives access to other Cardbox objects.

`Database`	A Database object that represents the database that contains these records.

Methods

`Add`	Adds a Record object to this collection.
`Remove`	Removes a Record object to this collection.
`Contains`	Checks whether a Record object is contained in this collection.
`Find`	Finds the position of a Record object in this collection.
`Keep`	Saves this collection as a kept selection, like Search > Keep > Keep.
`Tag`	Tags or untags the records in this collection.
`Print`	Prints records, like File > Print.
`WriteToFile`	Exports records to a file, like File > Export.

`WriteToString`	This behaves like `WriteToFile` but returns the exported text as a string instead of writing it to a file.
`ListIndexToFile`	This is the equivalent of pressing the Preview and Count button when making a selection: it finds matching index terms and writes the list to a file. If you want to preview all index entries and not just those that match a given collection of records, use the `ListIndexToFile` method of the Database object (p.129).
`ListIndex`	This behaves like `ListIndexToFile` but returns the list of matching index terms as a string instead of writing it to a file.
`Deduplicate`	Marks potential duplicate records, like Tools > Deduplicate.

Search and selection methods

`Select, SelectKept, SelectTagged, SelectData, SelectRecordNumber, SelectFromWindow, SelectFromRecords, SelectRelational, Exclude, ExcludeKept, ExcludeTagged, ExcludeData, ExcludeRecordNumber, ExcludeFromWindow, ExcludeFromRecords, ExcludeRelational, Include, IncludeKept, IncludeTagged, IncludeFromWindow, IncludeFromRecords, IncludeRelational`

The search and selection methods of the Window object change the visible selection in the object. The search and selection methods of the Records object don't do this: instead, they return a new Records object that contains the result of the selection, and the original Records object is left unchanged. No change occurs to the screen display, which means that these commands don't disturb the user and they don't waste time by needlessly updating the screen.

The Record object

Help Point 726 gives full details of how to get and use a Record object and includes links to articles on each property and method.
See also page 67.

The Record object represents a record in a database. You can use it to tag, untag, delete and undelete records, and to get to other objects that access the record's content. You can get hold of a Record object in various ways:

- The `ActiveRecord` property of a window.
- The `Item` property of a Records object.
- The `NewRecord` method of the Database object creates a Record object for a new, blank record.
- The `DuplicateRecord` method of the Record object creates a Record object for a new, separate record identical to the original.

Properties

`Deleted`	To delete a record, set this property to `True`.
`Tagged`	This property tells you whether a record is tagged. You can tag or untag the record by setting this property to `True` or `False`.
`UserEditing`	If the user is editing this record, this property will be `True`.
`Editing`	If this macro is editing this record, this property will be `True`.
`TextFormat`	You can use this to set the initial value of the `TextFormat` property of all Field objects retrieved from this Fields object. See page 124.

Object properties

The following properties give access to other Cardbox objects.

`Database`	A Database object that represents the database that contains this record.
`Fields`	A Fields object that contains Field objects for all the fields in this record, This is the way in which you access the actual text within a record.
`Images`	An Images object that contains Image objects for all the images in this record.

Methods

`Edit`	Starts editing this record.
`StartEditing`	Identical to `Edit`. It is here because some programming languages give a special meaning to the word Edit, which prevents it from being used as a method.
`Duplicate`	Creates a Record object representing a new record whose content is identical to the current record.
`Preload`	Tells Cardbox to retrieve a list of fields because the macro will be asking for them soon. This can make macros faster if they are dealing with a remote server: see Help Point 753.

Help Point 727 gives full details of how to get and use a Fields object and includes links to articles on each property and method.

The Fields object

The Fields object represents the fields in a record. You can get hold of a Fields object as follows:

- The `Fields` property of a Record object.
- The keyword `Fields` on its own in a macro will get the Fields object for the current record in the currently active window.

Properties

The Fields object has the standard properties shared by all collections:

`Count`	The number of Field objects in this collection.
`Item(`*n*`)`	The *n*th Field object in this collection. This is not normally a very useful way of accessing individual Field objects (unless you plan to access them all) because there is no guarantee as to the sequence in which the fields will be listed.
`Item("`*name*`")`	The Field object for the field whose name is *name*.
`Item("`*name1,name2,…*`")`	A compound Field object that represents several fields combined. This is more complicated to manipulate than a Field object for a single field, but it can be faster if you are retrieving or setting many fields in a database on a remote server. See Help Point 753.
`TextFormat`	You can use this to set the initial value of the `TextFormat` property of all Field objects retrieved from this Fields object. See page 124.

Method

`Preload`	Tells Cardbox to retrieve a list of fields because the macro will be asking for them soon. This can make macros faster if they are dealing with a remote server: see Help Point 753.

Help Point 728 gives full details of how to get and use a Field object and includes links to articles on each property and method.
See also page 68.

The Field object

The Field object represents a single field in a single record. You can use it to retrieve or set the text of a field. The only way to get hold of a Field object is through the Fields collection or, when the user is editing a record, through the `ActiveField` property of the Window object.

A special form of the Field object can represent more than one field at once. This is used when you need to handle many fields as fast as possible: see Help Point 753.

Properties

`Name`	The name of this field.
`Text`	The text of this field. In many cases you don't need to mention this property explicitly: see the examples on page 68.
`TextFormat`	This controls whether the `Text` property contains the bare text of the field, with no index information, or whether it also includes markers to show which words are indexed and which aren't: see page 70.
`TextLength`	The length of the text in this field, ignoring any index markers inserted by `TextFormat`.
`TextNoIndex`	The text of this field, with no markers inserted even if `TextFormat` has requested them.
`TextWithIndex`	The text of this field, with markers inserted to show which words are indexed and which aren't. This is an alternative to using `TextFormat`.

Object properties

The following property gives access to other Cardbox objects.

`Definition`	A FieldDefinition object that tells you the field's name and index mode.

Help Point 729 gives full details of how to get and use an Images object and includes links to articles on each property and method.
See also page 73.

The Images object

The Images object represents the images in a record. You can get hold of an Images object as follows:

- The `Images` property of a Record object.
- The keyword `Images` on its own in a macro will get the Images object for the current record in the currently active window.

Properties

The Images object has the standard properties shared by all collections:

`Count`	The number of Image objects in this collection.
`Item(`*n*`)`	The *n*th Image object in this collection.

Methods

`Add`	Adds an image to this collection.
`ReadFromFile`	Reads an image or images from a file.
`Paste`	Pastes an image from the Clipboard.
`MoveFromTo`	Rearranges the images in the list.

Help Point 730 gives full details of how to get and use an Image object and includes links to articles on each property and method.
See also page 74.

The Image object

The Image object represents a single image in a single record. The only way to get hold of an Image object is through the Images collection.

Properties

`Width`, `Height`	The width and height of this image, in pixels.
`IsNew`, `Type`, `NewWidth`, `NewHeight`, `Rotate`, `Compress`, `ResolutionX`, `ResolutionY`, `NewResolutionX`, `NewResolutionY`, `UpdateDisplayTextFormat`	These properties apply only to a newly added image before it is saved for the first time. They let you control how Cardbox processes and compresses the image.

Object properties

The following property gives access to other objects.

`Picture`	This property gives the visible image itself in a format that can be passed to other programs: eg. Visual Basic or Microsoft Office. The value of this property is a SharedPicture object: see Help Point 731.

Methods

`CopyToClipboard`	Copies this image to the Clipboard.
`WriteToFile`	Writes this image to a file.

The Databases object

Help Point 732 gives full details of how to get and use a Databases object and includes links to articles on each property and method. See also page 74.

The Databases object represents the databases that are currently open in this copy of Cardbox. You can get hold of a Databases object by retrieving the Databases property of the Application object.

Properties

The Databases object has the standard properties shared by all collections:

`Count`	The number of Database objects in this collection.
`Item(`*n*`)`	The *n*th Database object in this collection.
`Item("`*name*`")`	The Database object for the database whose name is *name*.

The Database object

Help Point 733 gives full details of how to get and use a Database object and includes links to articles on each property and method. See also page 75.

The Database object represents a Cardbox database as a whole. You can use it to retrieve objects or perform operations that are relevant to the whole database. You can get hold of a Database object:

- Through the `Item` property of a Records object.
- Through the `Database` property of a Window object.

Properties

`FullName`	The name of this database.
`Profile`	The name of the current profile.
`ShortUserName`	The short user name associated with the current profile.
`LongUserName`	The long user name associated with the current profile.

Object properties

The following properties give access to other Cardbox objects.

`Windows`	A Windows object that contains all windows that are using this database. Often there will be only one such window, and you can use `Windows(1)` to get its Window object.
`FieldDefinitions`	A FieldDefinitions object that contains the definitions of all the fields in the database.

`AllRecords`	A Records object that contains all the records in the database.
`NoRecords`	A Records object that contains nothing at all. This is a useful starting point when building up selections programmatically.
`TaggedRecords`	A Records object that contains all the tagged records in the database.

Methods

`SetProfile`	Selects a user profile.
`ClearKept`	Deletes a kept selection.
`ClearTagged`	Untags all records in the database.
`NewRecord`	Creates a new record in the database and returns its Record object.
`ReadFromFile`	Imports records from a file, like File > Import > From File.
`ReadFromCardbox`	Imports records from another window, like File > Import > From Cardbox.
`Download`	Downloads a database or format file from the server.
`Upload`	Uploads a database or format file to the server.
`ListIndexToFile`	This is the equivalent of pressing the Preview button when making a selection: it finds matching index terms and writes the list to a file. To emulate the Preview-and-Count button, use the `ListIndexToFile` property of the Records object (p.122).
`ListIndex`	This behaves like `ListIndexToFile` but returns the list of matching index terms as a string instead of writing it to a file.
`NextIndexTerm`	Finds the first, last, next or previous, indexed term for a field.

The FieldDefinitions object

Help Point 734 gives full details of how to get and use a FieldDefinitions object and includes links to articles on each property and method.

The FieldDefinitions object represents the definitions of the fields in a database. You can get hold of a FieldDefinitions object from the `FieldDefinitions` property of a Database object.

Properties

The Fields object has the standard properties shared by all collections:

`Count`	The number of FieldDefinition objects in this collection.
`Item(`*n*`)`	The *n*th FieldDefinition object in this collection.
`Item("`*name*`")`	The FieldDefinition object for the field whose name is *name*.

The FieldDefinition object

Help Point 735 gives full details of how to get and use a FieldDefinition object and includes links to articles on each property and method.

The FieldDefinition object represents the definition of a single field. You can get hold of it from:

- An item in the FieldDefinitions object.
- The `Definition` property of the Field object.

Properties

`Name`	The name of the field.
`Description`	The description of the field, entered when you designed the format file.
`IndexMode`	The index mode of the field: 0 = None, 1 = Manual, 2 = Automatic, 3 = All.
`Index`	The position of this field in the FieldDefinitions collection: the first field has `Index=1`.
`InternalNumber`	Cardbox's internal field number for this field. This is only needed in specialist applications, such as when you are handling data in the Cardbox internal dump format.

The Connection object

Help Point 736 gives full details of how to get and use a Connection object and includes links to articles on each property and method.

The Connection object represents a TCP/IP connection to an external computer. This is a technical object that is used for advanced purposes, such as sending email by direct use of the SMTP protocol.

Getting an Application object

In macros, `Connection` will create and return a Connection object.

This is the only way to get a Connection object. If you are using another programming language to drive Cardbox and want to set up a TCP/IP connection, you should use the TCP/IP facilities offered by that language.

Properties

`Timeout`	This property sets a time limit in milliseconds for the `Connect`, `ReadLine`, and `WriteLine` methods. If one of these methods takes longer than the limit, it is interrupted. The default value of this property is -1, meaning no limit.
`TimedOut`	This read-only property is `True` if the last `ReadLine` was interrupted because it exceeded the time limit.
`UTF8`	The default value of this property is `False`, meaning that text will be sent and received using the Windows character set. If you set it to `True`, the UTF-8 Unicode character set will be used instead.

Methods

`Connect`	Connects to a given IP address and port number.
`ReadLine`	Reads a line of text from the connection.
`WriteLine`	Writes a line of text to the connection.

Index of methods and properties

Help Point 700 is an electronic version of this list, with links to articles that describe each property and method in detail.

This list shows all methods and properties of Cardbox objects, in alphabetical order. It doesn't show any details of the way that they are to be used: to get more information, you have two choices:

- Look up Help Point 700 and follow the links to a full specification in the help file.
- Or, for methods that correspond to commands, press the Record button, perform the command yourself, and then look at what the macro recorder has recorded for you.

Note: All methods and properties marked with an asterisk (*) can be used directly from macros without any object reference. They then refer to the active window, the active record, etc.

Method or property	Object	Description
`Activate`	Application	Activates the Cardbox window.
`Activate`	Window*	Activates this window.
`ActiveField`	Window*	The field you are currently editing.
`ActiveFieldName`	Window*	The field you are currently editing.
`ActiveRecord`	Window*	The current record in this window.
`ActiveWindow`	Application*	The currently active database window.
`Add`	Images	Adds an image to the list.
`Add`	Records	Adds a record to this Records object.
`AddRecord`	Window*	Adds a new record and starts editing it.
`AllRecords`	Database	Gives the Records object that contains all the records in this database.
`Application`	All	Gives you the Application object.
`AutoPopup`	HistoryWindow	Checks or sets whether the History window automatically appears when a selection is changed.

Method or property	Object	Description
`Build`	Application	Information about this version of the Cardbox program.
`BuildNumber`	Application	Information about this version of the Cardbox program.
`Caption`	Application	The caption of the Cardbox window.
`Cascade`	Windows	Arranges the database windows in a cascade.
`CheckEditing`	Window*	Checks whether this window is currently editing a record.
`ClearKept`	Database	Deletes one of the kept selections for this database.
`ClearTagged`	Database*	Untags all records in this database.
`ClipboardText`	Application*	Checks or sets the text currently in the Windows Clipboard.
`Close`	Window*	Closes this window.
`Command`	Window*	When editing: this method performs a keystroke or a menu command.
`CommandLine`	(macros)	The command line passed to the macro when it was started.
`Compress`	Image	Controls how a newly added image is to be compressed.
`Contains`	Records	Checks whether a record is in this Records object.
`Context`	Application	The current context of the Cardbox user interface.
`CopyToClipboard`	Image	Copies the image to the Windows Clipboard.
`Count`	All collections	The number of objects in this collection.
`Database`	Record	The database that contains this record.

Method or property	Object	Description
`Database`	Records	The database that contains these records.
`Database`	Window*	The database that is open in this window.
`Databases`	Application*	The databases that are currently open.
`DateFromCardbox`	Application*	Converts a date from Cardbox format.
`DateToCardbox`	Application*	Converts a date to Cardbox format.
`Deduplicate`	Records	Searches for potential duplicate records and tags them.
`Definition`	Field	The definition of this field (name, index mode, etc).
`Deleted`	Record	Allows you to delete or undelete this record.
`DeleteRecord`	Window*	Deletes a record.
`Description`	FieldDefinition	The description of the field.
`Dial`	(macros)	Dials a telephone number.
`DoNotSaveRecord`	Window*	When editing: this method quits without saving the changes.
`Download`	Database	Downloads a database file or format file from the server.
`Duplicate`	Record	Creates a new record with content identical to this one.
`DuplicateRecord`	Window*	Adds a new record based on the current record and starts editing it.
`Edit`	Record	Edits this record.
`Editing`	Record	Checks whether you are editing this record.
`Editing`	Window*	Checks whether this window is currently editing a record.

Method or property	Object	Description
`EditRecord`	Window*	Starts editing a record.
`Exclude...`	Records	Exclude records from within this Records object and return a new Records object containing the remaining records.
`Exclude...`	Window*	These methods perform search commands within this window.
`ExtraText`	Window*	Checks or sets the display mode for extra text.
`FieldDefinitions`	Database	The definition of all the fields in this database.
`Fields`	Record*	The collection of fields within this record.
`Find`	Records	Finds a record in this Records object.
`Find`	Window*	Finds text in a record you are editing.
`FindNext`	Window*	Finds the next occurrence of the search text.
`FirstRecord`	Window*	Moves among the records in this window.
`Format`	Window*	Checks or sets the current format in this window.
`FullName`	Database	The name of this database.
`GetMailExchangers`	(macros)	Finds the computers that accept mail for a given email address.
`GoToField`	Window*	When editing: this method moves to a different field.
`GoToRecord`	Window*	Moves among the records in this window.
`Halt`	(macros)	Terminates the macro.

Method or property	Object	Description
`Height`	Application, HistoryWindow, Window*	Check or set the Cardbox window's size.
`Height`	Image	The size of this image, in pixels.
`HighlightMatches`	Window*	Checks or sets the mode for highlighting matching words after a search.
`HistoryWindow`	Application*	The object that represents the History window in Cardbox.
`Images`	Record*	The collection of images within this record.
`Include...`	Records	Select records from the database as a whole and return a new Records object that contains those records plus all the others within the original Records object.
`Include...`	Window*	These methods perform search commands within this window.
`Index`	FieldDefinition	The position of this definition in the FieldDefinitions collection (the first definition has Index=1).
`Index`	Window*	This window's position in the Window menu.
`IndexMode`	FieldDefinition	The index mode of the field.
`InternalNumber`	FieldDefinition	Cardbox's internal field number for this field. This is only needed in specialist applications, such as when you are handling data in the Cardbox internal dump format.
`IsNew`	Image	Says whether the image has been newly loaded.
`Item("name")`	Databases, FieldDefinitions, Fields, Windows	The item whose name is **"name"**.

Method or property	Object	Description
`Item(`*n*`)`	All collections	The *n*th object in this collection.
`Keep`	Records	Keeps the selection represented by this object (equivalent to the Search, Keep, Keep command).
`LastRecord`	Window*	Moves among the records in this window.
`Launch`	(macros)	Runs an external program or opens a document.
`Left`	Application, HistoryWindow, Window*	Check or set this window's position.
`ListIndex`, `ListIndexToFile`	Database, Records	Lists index terms in the database.
`LongUserName`	Database	The long user name associated with the current profile.
`MachineName`	Application	Identification of the computer that this copy of Cardbox is running on.
`MoveFromTo`	Images	Rearranges the order in which images are stored.
`Name`	Application	Information about this version of the Cardbox program.
`Name`	Field, FieldDefinition	The name of the field.
`Name`	Window*	The caption of this window.
`NewHeight`	Image	Controls how a newly added image is to be rescaled.
`NewRecord`	Database	Creates a new record in this database.
`NewResolutionX`, `NewResolutionY`	Image	Controls the resolution of a newly added image.
`NewWidth`	Image	Controls how a newly added image is to be rescaled.

Method or property	Object	Description
`NextIndexTerm`	Database	Finds the first, last, next or previous, indexed term for a field.
`NextRecord`	Window*	Moves among the records in this window.
`NoRecords`	Database	Creates a blank Records object which you can then add records to.
`NumberFromCardbox`	(macros)	Converts a number from Cardbox format.
`NumberToCardbox`	(macros)	Converts a number to Cardbox format.
`OpenFile`	Windows*	Opens a database file.
`Paste`	Images	Pastes an image from the Clipboard.
`Pause`	(macros)	Pauses the macro.
`Picture`	Image	The visible image itself, in a form that can be passed to other programs (eg. Visual Basic or Microsoft Office).
`Play`, `PlayText`	Application*	Plays a macro.
`Preload`	Fields, Record	This optional method tells Cardbox to retrieve some database fields because you will be using them later.
`Preload`	Image	This optional method tells Cardbox to start loading the image because you will later be using the Picture property or the CopyToClipboard or WriteToFile method.
`Print`	Records	Prints records.
`Print`	Window*	Prints records.
`Profile`	Database	The name of the current user profile.

Method or property	Object	Description
`ReadFromCardbox`	Database or Window*	Reads records from a Cardbox database
`ReadFromFile`	Database or Window*	Reads records from an external file
`ReadFromFile`	Images	Reads an image from an external file.
`RecordCount`	Window*	How many records there are in the current selection.
`RecordPosition`	Window*	Where you are in the current selection.
`Records`	Window*	The current selection of records in this window.
`Remove`	Records	Removes a record from this Records object.
`RemoveDeletedRecords`	Window*	Equivalent to the menu command View > Remove Deleted Records.
`ReplaceAll`	Window*	Finds and replaces all occurrences of a specified text.
`ReplaceHere`	Window*	Replaces one occurrence of a specified text.
`ResolutionX` `ResolutionY`	Image	The resolution of the image, in dots per inch.
`Rotate`	Image	Controls how a newly added image is to be rotated.
`Run`	(macros)	Runs an external program or opens a document.
`SafetyLevel`	(macros)	The current safety level of this macro.
`Save`	Record	Saves an edited or newly added record.
`SaveRecord`	Window*	When editing: this method saves the record.

Method or property	Object	Description
`SaveRegistryOptions`	Application	Saves Cardbox's section of the Windows Registry to a file.
`Scrap`	(macros)	Temporary storage values external to the macro.
`Select...`	Records	Select records from within this Records object and return a new Records object containing those records.
`Select...`	Window*	These methods perform search commands within this window.
`SelectionLevel`	Window*	The current level of selection.
`Sequence`	Window*	Checks or sets the current sequence of records in this window.
`SetFromRecords`	Window*	Set the current selection to the records listed in a Records object.
`SetOption`	Window*	When editing: sets a value in a check box, radio button, or drop-down list.
`SetProfile`	Database	Select a user profile.
`SetSequence`	Window*	Sets the current sequence of records in this window.
`ShortUserName`	Database	The short user name associated with the current profile.
`Sleep`	(macros)	Suspends the macro for a given time.
`SlotNumber`	Record	(this property is obsolete)
`StartEditing`	Record	Edits this record.
`Status...`	(macros)	These properties control the status display while a macro is running.
`StepBrowseMode`	Window*	Checks or cancels step browse mode for this window.
`StepBrowse...`	Window*	These methods start the step browse mode for this window.

Method or property	Object	Description
`Tag`	Records	Tags or untags a group of records.
`Tagged`	Record	Allows you to tag or untag this record or check its tagged status.
`TaggedRecords`	Database	Gives the Records object that contains all the tagged records in this database.
`Text`	Field	The text of this field.
`Text`	HistoryWindow	The text in the History window.
`TextLength`	Field	The length of the text in this field.
`TextFormat`	Field	Controls the format in which the `Text` property represents the text of the field.
`TextFormat`	Fields	Sets the default value of the `TextFormat` property of any Field object that you create.
`TextFormat`	Record	Sets the default value of the `TextFormat` property of any Fields object that you create.
`TextNoIndex`, `TextWithIndex`	Field	Versions of the `Text` property without, and with, index markers.
`Tile`	Windows	Tiles the database windows so they can all be seen at once.
`Timeout`	Connection	A time limit for the `Connect` and `ReadLine` methods.
`Top`	Application, HistoryWindow, Window*	Check or set this window's position.
`Type`	Image	The type of the image (Document or Picture).
`TypeText`	Window*	Types text into Cardbox at the current cursor position.
`TypeTextFromFile`	Window*	Type text into Cardbox from a text file.

Method or property	Object	Description
`UndeleteRecord`	Window*	Undeletes the last record that was deleted in this window.
`UndoOneSelectionLevel`	Window*	Undoes one or more levels of selection.
`UpdateDisplay`	Image	Redisplays the image after its properties have been changed.
`Upload`	Database	Uploads a database file or format file to the server.
`UserEditing`	Record	Checks whether the user is editing this record.
`UTF8`	Connection	Controls the character encoding used on this connection.
`Visible`	Application	Checks or sets the Cardbox window's visibility.
`Visible`	HistoryWindow	Checks or sets the History window's visibility.
`Visible`	Window*	Checks or sets the window's visibility.
`Width`	Application, HistoryWindow, Window*	Check or set the window's size.
`Width`	Image	The size of this image, in pixels.
`WindowNumber`	Window*	If several windows share the same name, which of them this one is.
`Windows`	Application*	The database windows that are currently open.
`Windows`	Database	The windows that have this database open.
`WindowState`	Application, Window*	Checks or sets this window's state (maximised, iconic, or normal).
`Workspace`	Application	The name of the workspace file.

Method or property	Object	Description
`WriteLog`	(macros)	Writes debugging information to the Cardbox log.
`WriteToFile`	Image	Writes the image to a file on disk.
`WriteToFile`	Records	Writes selected records to a file on disk.
`WriteToString`	Records	Like `WriteToFile`, but returns the written records as a string.
`ZOrder`	Window*	Where this window is relative to others.

You can find a map of the Cardbox object model on pages 58 and 110. The master index of methods and properties starts on page 132. A list of all examples and sample macros is on page 97.