EtCetera Help Index

This help file is for versions 1.25 (Shareware) and 2.10 (Retail) of EtCetera. The difference between the two is that 2.10 supports 386-only commands (with the ETCETERA.386 file) and 1.25 does not. For best results, maximize the Help window before reading this document.

Command Reference

EtCetera's commands are broken into functional groups.

Activate, Close, Hide, Maximize, Minimize, Restore, Unhide, Window
Alert, Ask, Message, Query
Align
ArrayBox
Beep, PlaySound
Bottom, Center, Left, Right, Top
Break, For, Next
Call
CD, ChDir, MD, MkDir, RD, RmDir
ChooseMenu
CommFileBox
Continue, ErrOff, Handle, OnErr, Retry
Copy, Del, Delete, Ren, Rename
CreateWindow, DestroyWindow
DDEExec, DDEPeek, DDEPoke
DisplayBitmap, PaintWindow, WriteText
End, Reboot, Shutdown, Stop
Execute, Run
FileFill, GetClipboard, ReadTo, SetClipboard, TitleFill, WriteFrom
Flash
FocusOn
GetConnection, NetConnect, NetDisconnect
GetDesktop, GetHandle, GetText
GetEntry, SetEntry
GetExec, SetExec
Goto
If/Then
Input
KeyOff, KeyOn, Toggle
Match
MDIAlign, MDICascade, MDIMaximize, MDIRestore, MDITile
Mouse
Move, Size
NextApp, PrevApp
Send, SendKeys, Type
SetAttr
Wait

Subject Index

CD, ChDir, MD, MkDir, RD, RmDir
Copy, Del, Delete, Ren, Rename
Execute, Run
FileFill, GetClipboard, ReadTo, SetClipboard, TitleFill, WriteFrom
GetEntry, SetEntry
Match
SetAttr

SetAttr

SetAttr is used to set file attributes. File attributes are information about the way a file can be accessed or has been accessed. There are five valid file attributes. NORMAL means that the file does not have any restrictions on its use. READONLY is a protective attribute - DOS will allow the file to be read but not written to or on top of; nor will DOS allow the file to be destroyed. This is easily circumvented, however, by restoring the attribute to NORMAL. SYSTEM and HIDDEN are two separate attributes which, for all intents and purposes, do nothing but keep the file from being viewed when you enter DIR at the DOS command line. The final attribute is ARCHIVE, which means that the file has been modified since the last time it was backed up.

The command's format is:

SetAttr [string] [attribute{s}]

[string] is the filename (which can be a pathname) whose attributes you want to change. You can specify the NORMAL attribute, or any combination of the other attributes. See the GetAttr keyword for information about determining a file's attributes.

Alert, Ask, Message, Query

These functions are used to display messages or ask questions from the user. The format of these commands are:

Alert [string] {string} {options}
Ask [variable] [string] {string} {options}
Message [string] {string} {options}
Query [variable] [string] {string} {options}

Alert and Message display a dialog box which contains a single OK button, plus text provided by [string]. Alert also displays an exclamation point icon. If {string} is specified, it becomes the title bar text for the dialog box. This is the default if no {options} are specified.

If any options are specified, then Alert and Message behave exactly the same, and their default behaviors are provided only for backwards compatibility with prior versions of EtCetera. The options you can specify are as follows:

OK	- displays just an OK button
OKCANCEL	- displays an OK and a Cancel button
YESNO	- displays a Yes and a No button
YESNOCANCEL	- displays three buttons: Yes, No, and Cancel
RETRYCANCEL	- displays a Retry and a Cancel button
ABORTRETRYIGNORE	- displays three buttons: Abort, Retry, and Ignore
INFO	- displays an information icon ("i" in a circle)
EXCLAMATION	- displays an exclamation point icon
QUESTION	- displays a question mark icon
STOP	- displays a stop-sign icon
NOICON	- displays no such icon (the default)
DEF1	- the first button is the default button
DEF2	- the second button is the default button
DEF3	- the third button is the default button

For both Alert and Message, which cannot return the user's selection to your program, only the OK button option makes sense. You can, however, display the other button options - it just does not matter which button the user presses, and your program will not be able to determine it.

The Ask and Query commands are used to retrieve data. Ask, by default, has Yes and No buttons. Query has OK and Cancel. Both display a question mark icon. Selecting any options will override this default behavior. The [variable] can be any kind of variable. Based on which kind you choose, the following values will be placed into the variable based on the button pressed by the user:

Button	String	Numeric
OK	"OK"	1
Cancel	"Cancel"2	
Abort	"Abort"	3
Retry	"Retry"	4
Ignore	"Ignore"	5
Yes	"Yes"	6
No	"No"	7

String means a string variable or string array variable. Numeric means any numeric variable - integer, integer array, long, or long array.

MDIAlign, MDICascade, MDIMaximize, MDIRestore, MDITile

These commands are used to manipulate MDI windows.   MDI is an abbreviation for "Multiple Document Interface."   It is a specification designed into Windows which simplifies having an application which supports having more than one document (spreadsheet, file, database, whatever) open at a time.

MDIMaximize and MDIRestore behave and are used exactly like their non-MDI counterparts, Maximize, and Restore, except that they only operate on the active MDI window.   MDIAlign will rearrange all MDI icons so that they are well-ordered at the bottom of the active window.   MDICascade will rearrange the MDI windows so that they are arranged in a folder-like fashion.   MDITile will instead rearrange the MDI windows so that they are all visible within the active window and none obstructs any other.

Note that these commands always affect the active application only.   This means that if Notepad is your active application, you cannot use these commands with Program Manager without first activating Program Manager.

Also be aware that many applications which support multiple documents do not properly conform to MDI conventions, so these commands should not be used with them (and will not generally work).   Microsoft's own applications (Excel, Word), for instance, do *not* follow these conventions.   You will have to send keystrokes to these applications to simulate the behavior you want.

See also FocusOn.

FocusOn

FocusOn is used to specifically activate a child window.   Note that, like the MDI commands, which operate only with true MDI applications, the FocusOn command may not operate properly with certain applications, particularly those which do not follow the standard MDI conventions.   You can, however, use SendKeys to send keystrokes (such as Ctrl+F6) to switch between child windows.   The actual keystrokes sent are entirely dependent upon the application involved; refer to the application's documentation for details.

See also the keyword ActiveChild.

GetKey

GetKey can be used to stop execution of EtCetera pending a keystroke from the user.   Its format is

GetKey [string variable]

The key name text is stored in the string variable you provide.   This key name text is completely dependent upon your keyboard and the driver used to control it under Windows.   This key name text is not, therefore, compatible with the SendKeys command.   Every key on your keyboard has a unique name; test the various values before relying on this command.

In general, key names for alphabetic keys are the same as the alphabetic character; function keys are represented by "F1", "F2", etc.; keys on the numeric keypad usually begin with "Num".

ArrayBox

The ArrayBox command is used to display a dialog box containing a list box. The list box is filled with entries from the string array structure. Its format is

ArrayBox [string] {string} [value] [value] [value] {value}

The first [string] is the descriptive text which will be placed into the dialog box. The {string} is optional and indicates the title bar text of the dialog box if included. The first two [value]s indicate the entries in the string array structure to place into the edit box. The first indicates the beginning string, and the second indicates the ending string. The third [value] indicates the entry in the string array structure with which to place the user's choices. The user can choose one or more items from the dialog box. The optional {value} indicates the last entry which can be used for the users responses. If it is not given, then it is assumed to be 999, the final entry of the string array structure.

For example,

ArrayBox "Which of the flavors do you like?", 101, 200, 201, 250

In this case, the strings from $[101] through $[200] will be placed into the list box of the dialog box. Then, once the user clicks OK, his choices will be placed into the string array structure starting with $[201]. If the user chooses three flavors, then $[201], $[202], and $[203] will contain his choices. If he chooses 60 flavors, then only 50 of his choices will be saved, since we limit the return entries to $[250].

You can determine how many choices the user made with the Lines keyword. If the user chooses Cancel, then Lines will instead be 0, and no modifications will be made to the string array structure.

ArrayBox automatically alphabetizes the list box.

ChooseMenu

ChooseMenu can be used to select an item directly off of an application's menu bar without having to send keystrokes [see the description of SendKeys] to the application window. Its format is

```
ChooseMenu [integer] [integer] {integer} ...
```

The first two [integer]s are required. The first one indicates which menu in the menu bar to use (such as File, Edit, Search, View, etc.). The second one indicates which item on the menu to choose. If the item chosen is a cascading menu [refer to your Windows User's Guide], then an additional {integer} is required. If a fourth level of menus is available, then another {integer} is required. Etc.

Menus are zero-based, so the first item on a menu is numbered 0. Also note that separators (lines which cross the menu) are items, so include them in your count.

For instance, if a menu bar contains two items, File and Edit, and the File menu has three items, Open, Save, and Exit, and the Edit menu has three items, Cut, Copy, and Paste, then to choose the File Save command, use

```
ChooseMenu 0, 1
```

and to choose the Edit Paste command, use

```
ChooseMenu 1, 2
```

CommFileBox

CommFileBox uses Microsoft's COMMDLG.DLL file to create a file list dialog box which is common among many of the Windows 3.10 "applets" (such as Notepad, Write, and Paintbrush). Thus, anyone who uses these programs will instantly recognize the intention of the dialog box.

The syntax of this command is:

CommFileBox [FILENAME [string variable] | PATHNAME [string variable]] [OPEN | SAVE] [string] {string}

CommFileBox can return either the simple filename of the user's selection, or the complete pathname, or both. To indicate which you want, use the keyword FILENAME or PATHNAME, followed by the variable into which you want the user's selection placed. You can use both FILENAME and PATHNAME - just be sure to follow each with a string variable.

You must specify either OPEN or SAVE to indicate which class of dialog box you want displayed. The SAVE keyword indicates that the dialog box should display the filenames in gray rather than in black, as with OPEN.

The required [string] is the title bar text of the dialog box. The optional {string} is the filename "filter" for the dialog box. The filter is a sequence of strings which appears in a drop-down list box ("combo box") and has associated with each string a certain file mask, such as *.TXT, *.EXE, *.PCX, and the like. Although it is usually of the form asterisk-dot-extension, it need not be.

The specify a filter, you must separate each item of the filter with a vertical bar character (|). For example, when you choose the File Open command in EtCetera, the combo box in the lower left corner of the dialog box displays

EtCetera Files (*.ETC)

and in the edit box in the upper left corner of the dialog box, you see

*.ETC

If you click on the down arrow of the combo box, you see

EtCetera Files (*.ETC)
All Files (*.*)

in the drop-down list box. If you click on All Files (*.*), then All Files (*.*) appears in the combo box, and *.* appears in the upper left edit box.

To set this up, the filter is setup as follows

EtCetera Files (*.ETC)|*.ETC|All Files (*.*)|*.*|

Note the vertical bar between each item. There are four items - two descriptions and two masks. Each mask must follow the description which describes it. There must also be a vertical bar at the end of the list.

So, for File Open, EtCetera itself uses something like:

CommFileBox Pathname $P, Filename $[100], Open, "EtCetera - Open File", "EtCetera Files (*.ETC)|*.ETC|All Files (*.*)|*.*|"

Then $P is used to open the file, and $[100] is used to display the name of the file in EtCetera's title bar.

CreateWindow, DestroyWindow

CreateWindow can be used to create a generic window on which you can display text or bitmaps. DestroyWindow eliminates a window so created.

Their formats are:

```
CreateWindow [handle] [integer] [integer] [integer] [integer]
DestroyWindow [handle]
```

The [integer]s for CreateWindow are, respectively, the horizontal position, the vertical position, the width, and the height of the window. Horizontal position and vertical position are relative to the upper left corner of the video display and indicate the position of the upper left corner of the window. All values are in pixels.

DestroyWindow is used to get rid of a window displayed by CreateWindow. It should not be used to eliminate windows which have not been created with CreateWindow. [You can get UAEs or GPFs by doing so.]

See DisplayBitmap, PaintWindow, WriteText for information on how to use these windows.

DDEExec, DDEPeek, DDEPoke

The DDE commands can be used to control Windows applications or to communicate with them. In order for these commands to be available, you must have the file DDEML.DLL in your Windows System subdirectory. This file is included with EtCetera for Windows 3.00 users; for Windows 3.10 users, this file is already provided.

DDEExec causes EtCetera to request that a DDE command be executed by another application. Any commands supported by other applications are completely dependent upon those applications, so no data can be given here.

The format is

```
DDEExec [string] [string] [string]
```

The first string is the DDE name of the application (which is not always the name of the program itself - refer to the documentation of your other application). The second string is the "topic" name. This is usually the name of the document (spreadsheet, database, etc) of the application. The third string is the actual command itself.

DDEPeek and DDEPoke are used to retrieve data from and send data to DDE applications, respectively.

```
DDEPeek [string] [string] [string] [string variable]
DDEPoke [string] [string] [string] [string]
```

For both commands, the first three [string] parameters are the same. The first is the name of the DDE application. The second is the topic. The third is the actual item whose data you want to retrieve or set. For DDEPeek, the [string variable] is the variable where you want the data retrieved stored. If the data exceeds 255 characters, it will be truncated.

For DDEPoke, this is the actual text you want sent to the application. If you want to send numeric data, translate it to a string first.

DisplayBitmap, PaintWindow, WriteText

These functions are generally used to alter windows created with the CreateWindow command, although they can be used on any general window type.

DisplayBitmap can take a standard bitmap file created in PaintBrush and display it in a window. The format is:

DisplayBitmap [handle] [string] [integer] [integer]

[handle] is the window handle (from CreateWindow, GetDesktop, or GetHandle), [string] is the name of the bitmap file, and the [integer]s are the horizontal position and vertical position, respectively, to place the bitmap with respect to the upper left corner of the window.

PaintWindow will fill a window with the color specified. Its format is:

PaintWindow [handle] [color text]

[color text] is any one of the following words: BLACK, WHITE, GRAY (or GREY), RED, ORANGE, YELLOW, GREEN, BLUE, PURPLE. [color text] is not case-sensitive.

WriteText can be used to display written data on a window. Its format is

WriteText [handle] [string] [integer] [integer]

[string] is the text to write (which can be any character which Windows is capable of displaying), and the [integer]s behave exactly as they do with DisplayBitmap.

GetConnection, NetConnect, NetDisconnect

These functions deal with establishing or eliminating links on local area networks (LANs) such as Novell Netware, Banyan Vines, Artisoft's Lantastic, or any other network which supports Windows (and which Windows supports).

NetConnect allows you to establish a connection with a remote device, such as a printer or file server.   It does not allow you to log into the network.   Its syntax is

NetConnect [string] [string] {string}

The first [string] is the name of the device as you would call it from your workstation, such as F: for a disk device or LPT2: for a printer device.   The second [string] is the network's name for the device, such as \\TOADPIPE\SYSTEM or ADMIN/MAIL.   This is completely dependent upon your network software, and you should consult its documentation for more information.

The optional {string} is a password, if required by your network.   Refer to your LAN documentation for details.

NetDisconnect reverses the process, eliminating a previously established link.   Note that you can disconnect any network device with this command, not just devices connected with the NetConnect command.   Its format is:

NetDisconnect [string]

where [string] is the local device name, such as F: or LPT2:.

GetConnection will return the network name of a connected device.   Its format is

GetConnection [string] [string variable]

[string] is the local name, and [string variable] is the variable where you want the device's network name to be stored.   For example,

GetConnection "LPT1:", $N

would copy the network name of the printer into $N, such as "MARX/HARPO".   The names are, of course, completely at the mercy of the sense of humor of your network administrator.

GetDesktop, GetHandle, GetText

These functions retrieve information related to handles.

GetDesktop returns a handle for the desktop, the background upon which all other windows are drawn.

GetDesktop [handle]

GetHandle returns a handle to a window whose title bar is uniquely identified by the string provided:

GetHandle [handle] [string]

GetText reverses the process and returns the title bar text, if any, of the window specified by [handle]:

GetText [handle] [string]

If/Then

The If/Then construction can be used to evaluate various circumstances while you EtCetera batch file executes. The general format is:

```
If [condition] {THEN} [command...]
```

The condition can be one of two things. The first is a comparison between two like items, such as strings or numeric values. To compare strings (which can be literal strings, string variables, or string array members), you use the following operators:

= or EC	tests for exact equality (case-sensitive)
<> or NC	tests for exact inequality (case-sensitive)
EI	tests for equality, case-insensitive
NI	tests for inequality, case-insensitive

For example,

```
If $A = "Cancel" Then Goto EndOfProgram
If $A EC "Cancel" Goto EndOfProgram
```

These test to see if $A contains "Cancel", exactly, and, if so, transfer execution to the command following the label EndOfProgram. [See the description of the Goto command for more information.] The use of "Then" is optional.

To compare numeric values (which can be integers, longs, or either array type), use the following operators:

= or EQ	tests for equality
<> or NE	tests for inequality
< or LT	tests for "less than"
> or GT	tests for "greater than"
<= or LE	tests for "less than or equal to"
>= or GE	tests for "greater than or equal to"

For instance,

```
If ![12] <= 14 Then Message "![12] is less than 15"
If ![12] LE 14 Then Goto Failure
```

These both test ![12] to see if it is less than or equal to 14 (which is effectively the same as less than 15) and, if so, displays a Message dialog box or transfers program flow to the label Failure.

The second condition is used simply check something. With numbers, [condition] is true if the number is non-zero; [condition] is false if the number is 0. With strings, [condition] is true if the string contains anything; [condition] is false if the string is empty.

This can be used as follows:

```
If IsMin "Notepad" Then Maximize "Notepad"
IF $A THEN $[13] = UPPERCASE $A
```

See IsMin, Maximize, and UpperCase for a description.

Execute, Run

Execute and Run (which behave exactly the same) execute a program. The syntax is

```
Execute [string] {how}
Run [string] {how}
```

[string] is the command line used to run a program. This should be the name of the program file plus any required parameters, such as

```
Run "NOTEPAD WIN.INI"
```

This line would run Notepad and, because Notepad interprets the command line, Notepad would load the WIN.INI file. Note that the extension (.EXE) is not required.

The option {how} parameter is either MAXIMIZED (MAX), MINIMIZED (MIN), or WINDOWED (WIN), to tell EtCetera how to display the program initially. For instance,

```
Execute "NOTEPAD.EXE", MIN
```

would bring up Notepad as an icon. Keep in mind that running an application also makes it the active application, even if it is an icon.

Mouse

Mouse allows you to simulate mouse activities. It has two formats. The first format is

```
Mouse MOVE [integer] [integer]
```

This form moves the mouse to the screen coordinates indicated. The upper left corner of the display is (0, 0).

The second format is

```
Mouse [button] [what]
```

This format allows you to simulate mouse clicks. [button] is either LEFT, RIGHT, or MIDDLE, indicating which button you want to click. [what] indicates what you want to do with the button. It can be DOWN, UP, CLICK, or DCLICK. DOWN acts as if the button has been pressed (but not released). UP simulates letting go of the button. CLICK both presses and releases the button. DCLICK double-clicks the button, simulating two rapid clicks.

You can simulate drag-and-drop with EtCetera by using this command. For example, assuming that, in Program Manager, my Control Panel icon is located in the vicinity of (80, 80), and I want to move it to a different group, at position (240, 160), then the following code fragment would suffice:

```
Mouse MOVE 80, 80
Mouse LEFT DOWN
Mouse MOVE 240, 160
Mouse LEFT UP
```

Remember that the coordinates given are always with respect to the upper left corner of the display and, as such, care should be taken to ensure that objects are located where you think they are. Otherwise, the use of this command lends itself to peculiar results.

Bottom, Center, Left, Right, Top

The commands move the active window (which must be a window, and neither an icon nor maximized) to the corresponding position on the display. Top moves the window up to the top of the display window (with no horizontal motion), and Bottom moves the window to the bottom of the display. Left and Right move the active window to the left and right sides of the display, with no vertical motion. Center centers the active window on the display.

None of these commands use any parameters.

FileFill, GetClipboard, ReadFrom, SetClipboard, TitleFill, WriteTo

These commands use the string array structure (variables accessed $[x] - see the description of strings for more details). TitleFill copies the title bar text of all open applications into the string array. GetClipboard reads all of the text data (assuming some text data is available) from the clipboard and places it into the string array. SetClipboard writes data from the string array to the clipboard, destroying whatever was there.

In each of these cases, the format of the command is

```
[command] [integer] {integer}
```

where [command] is the respective command, [integer] is a numeric value between 0 and 999 to indicate the beginning position in the string array to use, and {integer} is the last position to use in the string array. [Note that this behavior differs from previous versions.] If the last position is not specified, then it is assumed to be 999.

For example,

```
GetClipboard 201, 220
```

retrieves the first 20 lines of text from the clipboard and places it into string array items 201 through 220. If there are fewer than 20 lines of text in the clipboard, then the remaining string array entries are unchanged.

You can use the Lines keyword to get the number of string array entries actually changed by any of these commands.

ReadFrom, WriteTo, and FileFill are similar to the others, but they take an additional parameter:

```
[command] [string] [integer] {integer}
```

The [string] parameter is a filename for ReadFrom and WriteTo, and a file mask (including wild card characters * and/or ?) for FileFill.

ReadFrom reads lines from a file into the string array structure starting with entry [integer] and ending at entry {integer}. Again, if {integer} is not specified, it is assumed to be 999.

WriteTo reverses the process and writes data from the string array structure into a file. This functions creates the file if it does not exist, or destroys the previous file if it does exist, and then writes the lines to the file.

FileFill fills the string array structure with files which match the description given by [string]. For instance,

```
FileFill "*.EXE", 351, 400
```

fills the string array structure starting with position 351 with all files which end in the extension EXE, up to entry 400 (if 50 or more files in the current directory end in the extension EXE). The Lines keyword will return the number of string array items actually changed.

Move, Size

Size and Move change the position or size of the active window. If the active window is an icon or is maximized, then Size and Move do nothing.

Size [integer] [integer]
Move [integer] [integer]

The first [integer] is the horizontal position or width. The second integer is the vertical position or height. Both of these numbers are in terms of pixels, individual screen points. The upper left corner of the display is always coordinate (0, 0). The lower left corner of a standard VGA-type display, for example, is (639, 479), since a VGA display is made up of 640 pixels horizontally and 480 pixels vertically.

You can use the SCRW and SCRH keywords to get the width and height of the display window in pixels. Width and Height return the width and height of the active window. XPos and YPos return the horizontal (X) and vertical (Y) positions of the active window.

Activate, Close, Hide, Maximize, Minimize, Restore, Unhide, Window

These window functions are pretty self explanatory. Their format is

[command] {string}

-or-

[command] {handle}

{string} is the unique title bar text of the window, or {handle} is the handle of the window, on which the command is to operate. If neither {string} nor {handle} is specified, the command will operate on the active window. Note that

Activate

is a do-nothing command, which activates the active window. The active window is the one which receives user input, such as keystrokes or mouse actions. It is also generally, by default, the target of most of the things EtCetera does.

Close shuts down an application, closing its window(s). Hide makes a window disappear (but the application continues to run). Unhide restores a window so that it is no longer hidden. Maximize maximizes a window. Minimize changes the window to its iconic state. Window takes a maximized window or an icon and makes it a standard window. Restore takes an icon (only) and returns it to the state previously held immediately prior to becoming an icon - either a standard window, or a maximized window.

The unique title bar text is enough text of the title bar so as to uniquely identify the window. For instance, you can run Notepad more than once. Assume that in one copy (instance), you have loaded WIN.INI, and in the other, you have loaded AUTOEXEC.BAT. This means that the title bars of each instance of Notepad will be "Notepad - WIN.INI" and "Notepad - AUTOEXEC.BAT", respectively. To close the instance of Notepad with the WIN.INI, you need to use the command

Close "Notepad - W"

which is enough of the title bar text to uniquely identify which instance of Notepad you want to close. If you were to use the command

Close "Notepad"

there is no guarantee which instance will be closed (although certainly one instance will be closed).

You can, if you want, use the entirety of the title bar text:

Close "Notepad - WIN.INI"

But if you know you have only one instance open, it is much easier to use

Close "Note"

so long as no two windows have those same four characters at the beginning.

The title bar text is case-sensitive.

In addition, for example, if you have retrieved a handle to Notepad with

```
GetHandle @N, "Notepad -"
```

you could also use

```
Close @N
```

to close Notepad.

See the description of GetHandle for details.

CD, ChDir, MD, MkDir, RD, RmDir

These directory-manipulation routines behave much like they do under DOS. CD and ChDir change the active directory. Note that, unlike DOS, you can (and must) use CD and ChDir to change the active drive.

MD and MkDir are used to create (make) a directory. RD and RmDir destroy (remove) a directory. Like DOS, the directory must be empty before it can be removed.

You can use literal text or any string with these commands. For example,

```
CD C:\WINDOWS
CD "C:\WINDOWS"
```

are the same. This allows you to use string variables to create, remove, and change to directories.

Copy, Del, Delete, Ren, Rename

These commands behave just like they do with DOS. Delete and Rename behave exactly like Del and Ren, respectively.

Copy copies files. Its format is

```
Copy [source] {destination}
```

[source] is the DOS filename, partial pathname, or complete pathname of the file(s) you want to copy. It can contain wildcard characters. {destination}, if specified, is where you want the files copied. If can be within the same directory (with a different filename), to a different directory, or to another drive. If {destination} is not specified, it is assumed to be the current directory with the same file names as the source. You can use either literal text or any string.

```
Copy C:\BACKUP\*.* D:\BACKUP
Copy "A:\*.DLL" SysDirectory
```

See the description of the SysDirectory keyword for more information.

Del and Delete delete files. The format is

```
Del [file mask]
Delete [file mask]
```

where [file mask] is a filename, partial pathname, or complete pathname, including possible wild card characters, indicating the file(s) you want to delete.

Ren and Rename can be used to rename files. Their format is

```
Ren [source] [new name]
Rename [source] [new name]
```

Examples:

```
Rename "*.BAT" "*.BAK"
Ren *.TXT "*.BKP"
```

Although [source] can be any valid DOS filename, partial pathname, or full pathname, including wild card characters, [new name] can only be a filename, including wild card characters, since rename changes the name of the file(s), but does not actually move it to another directory. Therefore, a pathname would be irrelevant.

The number of files affected by any of these commands can be determined by checking the value of the Files keyword.

End, Reboot, ShutDown, Stop

386 Reboot, ShutDown

The End command immediately stops execution of your program. Stop also stops execution of the program, but it automatically displays a dialog box with a stop-sign icon and an OK button. It uses the exact same syntax as the Message command, so you can replace the stop-sign icon. See the Ask, Alert, Message, Query topic for more information.

Neither of these commands are required to end a program - if there are no more lines of code to execute, EtCetera automatically stops. But you can use End to stop the program somewhere in the middle, if desired. Stop is generally for debugging purposes, allowing you to display program codes or other data while also ending the program. Using Stop is like using Message followed by End, but it takes only one command instead of two.

ShutDown also terminates an EtCetera program, but it also attempts to shut down Windows. The behavior of this command varies. If you are using EtCetera 1.25, then the following applies. If a DOS application is running, this will fail. If another Windows program needs to save data, the other application will prompt the user to do so. If the user tells the other application to cancel the shutdown (which most applications support), then Windows will not terminate - even though EtCetera will. Therefore, take these situations into account if you want to use this command. The format is

```
Shutdown {integer}
```

where {integer} is an optional return code which can be retrieved with the DOS ERRORLEVEL keyword if you run Windows from a DOS batch file. Refer to your DOS documentation for more information.

If, instead, you are running EtCetera 2.10 with ETCETERA.386 and Windows is in 386 enhanced mode, then ShutDown occurs immediately and always returns to DOS, regardless of what is running. Your Windows applications will not have the opportunity to save any unsaved data.

The Reboot command, specific to EtCetera 2.10 with ETCETERA.386, reboots the PC immediately.

NextApp, PrevApp

PrevApp and NextApp are used to activate a window indirectly. Windows, internally, keeps a list of windows, with the active window at the top of the list, the most recently active window as the next item on the list, and, at the bottom of the list, the window which has not been activated longer than any other.

NextApp activates the second window listed in the internal windows list (which was the active window prior to the active window being the active window). PrevApp activates the window at the bottom of the window list.

What this means is that NextApp can be used to ping-pong between two applications. When you use NextApp, the second window in the list becomes active, which places it at the top of the list, and the active window goes to position #2 on the list. Using NextApp again reverses the process.

PrevApp, on the other hand, can be used to round-robin through all open applications. This takes the window at the bottom of the list and places it on the top, effectively pushing all other windows down one position in the list. Doing this repeatedly will eventually activate every window.

Note that you cannot activate a hidden window.

GetEntry, SetEntry

These commands allow you to view and modify entries in initialization files, such as WIN.INI, SYSTEM.INI, or any other such file. Their formats are:

```
GetEntry [section] [entry] [string variable] {filename}
SetEntry [section] [entry] [string] {filename}
```

GetEntry retrieves an entry, and SetEntry sets it.

[section] is a string indicating the section name in the initialization file. A section is an area of the file with related entries, headed by text surrounded by square brackets [], such as [windows], [devices], or [386enh]. You do not use the square brackets when you indicate the section in GetEntry and SetEntry. [entry] is a string indicating the entry name. An entry name is followed by an equal sign, and the text following the equal sign is the value of the entry. [Do not include the equal sign when indicating the entry.] That value is what is specified by [string variable] and [string] above. GetEntry requires a string variable, since it has to have someplace to put the retrieved value. SetEntry can take any string (including an empty one) to change an entry.

These commands modify the file instantly and also make Windows aware of the fact of the change. This means that you can modify entries for programs on-the-fly, prior to running an application, without having to restart Windows (as you would have to do if you were to modify the WIN.INI file using Notepad). This will not, however, affect settings which Windows uses for initialization. For instance, you cannot use this to install drivers instantaneously by writing new entries to the SYSTEM.INI file. The data you write will be added, but it will have no effect on Windows until you restart it.

If you do not specify {filename}, it is assumed to be the WIN.INI file.

For example,

```
GetEntry "windows", "device", $D
SetEntry "extensions", "etc", "etcetera.exe ^.etc"
```

Send, SendKeys, Type

386 Type

Send, and SendKeys are synonyms of the same command. Each sends keystrokes to the active window. Note, however, that they do not behave as expected with DOS applications. Type, which requires that ETCETERA.386 is loaded and that you run Windows in 386 enhanced mode, can be used to send keystrokes to full-screen DOS applications.

The format of these commands is

```
[command] [string]
```

where [command] is either Send, SendKeys, or Type, and [string] represents the keystrokes you want to send to the active window.

To send text, use

```
SendKeys "This is sample text."
```

To send any non-text keys, place the keytext in curly braces. For example, to press the Enter key, use

```
SendKeys "{Enter}"
```

Both text and non-text keystrokes can be included in the same command, so the above two examples could be combined as

```
SendKeys "This is sample text.{Enter}"
```

Function keys are also represented by including the key name in curly braces, such as

```
SendKeys "{F1}"
```

You can also send Alt, Ctrl, and Shift keystrokes. To indicate one of these keys, use /a, /c, or /s, respectively. For example, to send Ctrl+Alt+F4, use

```
SendKeys "/c/a{F4}"
```

To send a "/" character, use it twice:

```
SendKeys "//"
```

To send an actual curly brace, precede it with a slash:

```
SendKeys "/{"
```

Keypad keys can be sent by using "{KP_}", where the "_" represents the key. For example, the keypad version of "9" can be sent by using

```
SendKeys "{KP9}"
```

This also works for non-numeric keys: {KP+}, {KP-}, {KP*}, etc.

The typical 101-key keyboard has two copies of certain keys, called enhanced keys. These include the right Ctrl and Alt keys, the inverted T arrow keys (between the main keyboard and the keypad), and the six keys which make up the edit cluster directly above the inverted arrow keys. (The "unenhanced" version of most of these keys is available on the keypad when the NumLock key is off.)

You can specify these keys by using the "/e" prefix. Note that this prefix only "enhances" that which immediately follows it. So, since there are "enhanced" versions of the Ctrl and Alt keys, the following two commands result in different meanings:

```
SendKeys "/e/c{Del}"
SendKeys "/c/e{Del}"
```

The first means "hold down the right Ctrl key and press the Del key on the keypad". The second means "hold down the left Ctrl key and press the Del key on the edit cluster". Some programs specifically use the "same" key differently (standard versus enhanced), so this function allows you to access each individually.

Since you can apply "/e" to any key, you can create key combinations which do not exist.

Also, be aware that Type enters keystrokes just as if the user had typed them in. Therefore, the states of the NumLock and CapsLock keys affect what will be displayed. You can use the KeyOn, KeyOff, and Toggle commands to modify the state of these keys prior to using Type.

Continue, ErrOff, Handle, OnErr, Retry

These commands are used to handle errors which occur with EtCetera. The first one you will use is OnErr, which is used to tell EtCetera to transfer control to a specific line (label) when an error occurs. In this way, you can prevent the default error dialog boxes from appearing and instead do something to correct the error.

The format of OnErr is

OnErr [label]

just like the Goto command. Then, whenever an error occurs, EtCetera will go to the line whose label is given. Once there, you can use the ErrVal keyword to determine which error actually occurred.

Once an error occurs and EtCetera has jumped to your error handling routine, you can do anything you like.

EtCetera also has some commands available which allow you to recover from the error situation. The first is the Continue command, which tells EtCetera to ignore the line where the error occurred and instead just continue with the line following the one where the error occurred. If you want EtCetera to try to execute the offending line again, you can use the Retry command. Usually, you will do something before using the Retry command, such as prompting the user for the location of a file or requesting that the user retype some title bar text. And if, instead, you want EtCetera to go ahead with its default behavior for the error (display a dialog box reporting the error), then use the Handle command. You will usually do this when an error occurs which you do not want to otherwise handle on your own. A good example of such as error is the "unknown and/or unrecoverable" error, described in the section of this help file entitled Errors Reference.

To disable the error trapping and return EtCetera to its default behavior, use the ErrOff command.

GetExec, SetExec

386 GetExec, SetExec

The GetExec and SetExec commands are specific to EtCetera 2.10 with ETCETERA.386. These commands allow you to determine and modify the execution priorities of any DOS application which is running, or of Windows itself. By execution priorities is meant the Foreground and Background priorities of an application, as well as which operating mode(s) it uses: Background, Exclusive, and/or High-Priority Background mode. Refer to the description of PIF Editor in your Windows User's Guide for information on these terms. The only term you will not find is High-Priority Background mode.

High-Priority Background execution is an operating mode which is built into Windows but is generally unavailable to users. By setting this mode, you guarantee that a DOS application will continue to run in the background, even if an exclusive mode application is running. You can also set Windows to operate this way, but Windows itself (including your Windows applications - all of which behave as a single unit) will not operate in this mode. Therefore, if you set Windows to High-Priority Background and then switch to a DOS application and set it to Exclusive mode, EtCetera (and all of your Windows applications) will stop running.

You can use SetExec as follows:

SetExec [value] {value} {value}

The first value is the foreground priority and is required. The second value is the background priority and is optional unless you want to indicate the execution mode(s). The third value indicates the specific execution modes and is optional. In order to specify the execution modes, you must also specify the background priority.

The priorities are just like those indicated in the Windows User's Guide and should be between 1 and 10000. You can use 0 to tell EtCetera not to modify the setting (i.e. leave it alone). The execution modes are indicated as the sum of the following values:

1	Exclusive
2	Background
512	High-Priority Background

If you want to turn on all modes, you would use 515. If you want to turn on exclusive only, use 1.

GetExec can retrieve the values:

GetExec [numeric variable] {numeric variable} {numeric variable}

EtCetera will store the foreground priority in the first variable, the background priority in the second variable, and the execution mode code in the third variable. The second and third variables are optional.

Goto

The Goto command transfers the flow of execution from one location to another. The location must be specified with a label. A label begins with a colon, contains any text characters, and ends with a space.

For instance, take the following code fragment:

```
:Begin
Match $A, "*.EXE"
If $A = "" Then Goto NoFiles
Ask $B, "Found file: " & $A & ".   Use this file?"
If $B = "No" Then Goto Begin
Goto GotFile
:NoFiles
Message "No more files."
End
:GotFile
...
```

This program starts by checking the active directory for a file whose extension is EXE. If it does not locate one, then it transfers control to the line following the label :NoFiles. If it does find one, then it presents the user with its name in a dialog box, asking if the user wants to use the located file. If the user chooses the "No" button, then the program jumps back to the line after the label :Begin, to see if there are any other files with the EXE extension. If the user chooses "Yes" (the only other option for a default Ask command), then the program jumps to the line following the label :GotFile, which is illustrated here with "...". The program would at this point do whatever it needs to do with the file the user selected.

Labels are case-sensitive. Labels cannot be keywords, although they can be commands. In general, however, if a word is defined in EtCetera, it is recommended that it not be used as a label. For instance,

```
Goto End
```

means that there is a label :End somewhere. End is a command in EtCetera and, although you can use it, it is not recommended. However,

```
Goto Height
```

will fail. Height is a keyword, and EtCetera translates it.

Wait

Wait's purpose is pretty self-explanatory. It has a variety of uses.

If Wait is used with no parameters, EtCetera will display a dialog box, asking the user to click the OK button before continuing the program.

You can also follow Wait with an integer, such as

```
Wait 3
Wait 3 seconds
```

meaning that it should wait three seconds before continuing. You can also following it with the word "second" or "seconds" if you want. For that matter, you can follow it with the word "minutes" or "hours", but it always assumes seconds.

Be aware that if you are running in 386-enhanced mode and you have a DOS application running full-screen, Wait may behave strangely. Of course, if the DOS application is running in Exclusive mode, EtCetera will stop running while the DOS application is running. But if the DOS application is not in exclusive mode, EtCetera will continue to run. Wait, however, suffers from a problem documented in Microsoft's own database which causes Windows timer routines to operate at up to 16 times slower than they should. We have attempted to compensate for this problem, but there can be a relatively large variance (within 3 seconds).

Another use is

```
Wait Closure
```

This form waits until the active window is closed (by the user). This allows you to permit the user to do something, like enter data into a dialog box in some application and then close it by clicking OK, before going on. The active window can be any window - an application or a dialog box.

The final alternative of this command is:

```
Wait Until [string] [what]
```

where [string] is the unique title bar text (see the description of the Activate command for details) of a window and [what] describes which event is desired. These can be either CLOSES, OPENS, MINIMIZES, MAXIMIZES, or WINDOWS. For instance

```
Wait Until "Notepad" Opens
```

causes EtCetera to wait until it can locate a window whose title bar text begins with "Notepad". If such a window already exists, then EtCetera continues right away.

Likewise, CLOSES waits until the first window it locates with the title bar text specified is closed (much like Wait Closure, except Closure is for the active window only). MINIMIZES, MAXIMIZES, and WINDOWS waits until the corresponding condition is met for the window in question. With CLOSES, WINDOWS, MINIMIZES, and MAXIMIZES, if no window with the specified title bar text exists, EtCetera continues execution right away. You can use the IsOpen keyword to check for the presence of a window prior to using any of these options.

Input

The Input command can be used to retrieve text data from the user. Input displays a dialog box with an edit box and OK and Cancel buttons. The user can enter text into the edit box and then press Enter (click OK), in which case the text the user enters will be placed into a specified string variable. If the user clicks Cancel or presses Esc, then the string variable will be made blank.

The format is

```
Input [string variable] [string] {string} {string}
```

[string variable] is the variable into which the user's entry will be placed. [string] is the descriptive text for the dialog box - a question, a request, or whatever. The first {string} is the default value - this will be displayed in the edit box when the dialog box is first displayed. The user can change this if desired. The second {string} is the title bar text of the dialog box. If you want to specify the title bar text, you must specify a default value. If you want the edit box's default value to be blank, use "".

For example:

```
Input $[104] "Please enter your name:", "", "Logon"
```

Beep, PlaySound

PlaySound plays a wave file. A wave file is a digital recording of sound stored in disk in a file, usually with the extension WAV. In order to do this, you must have a Windows-compatible sound card and/or sound driver, and Windows 3.10 or higher, or Windows 3.00 with the Multimedia extensions. If you do not have a sound card, this command becomes a do-nothing command. Its format is

```
PlaySound {string} {options}
```

{string} represents a wave filename, partial pathname, or complete pathname. The options are SYNC, ASYNC, LOOP, and OFF. {string} is required for all options except OFF. SYNC tells EtCetera to wait until the wave file has completed playing before continuing. ASYNC tells EtCetera to start the file playing and then continue execution immediately. This feature works only if your sound hardware supports it.

The LOOP feature tells EtCetera to set the sound hardware to continuously play the sound file until explicitly told to stop. You can stop an ASYNC or LOOP wave file by using PlaySound a second time with a new file and setting, or use the OFF option, which terminates any sound in play.

Examples:

```
PlaySound "CHIMES.WAV", Loop
PlaySound "JFK3.WAVE", Async
PlaySound OFF
```

Beep beeps the system speaker, or plays the default beep sound, depending upon your Windows configuration. Beep takes an integer parameter, which indicates how many times to beep. With no parameter, it defaults to once.

Examples:

```
Beep
Beep 3
Beep 4 times
```

Break, For, Next

This series of commands is used to execute a piece of your program repeatedly. You set it up by setting a numeric variable to a start value with the For statement, then execute some code. Once the code you want to repeat is done, you use the Next command, which tells EtCetera to increment or decrement the variable by the amount you specified in the For command, then transfer control to the line following the original For command. This continues until the variable is either greater than or less then the stop value specified with the For statement, depending on whether or not EtCetera is incrementing or decrementing the value.

For's format is:

```
For [numeric variable] {=} [value] {TO} [value] {STEP} {value}
```

For example (pun intended):

```
For #A = 1 to 19 Step 2
Message $#A
Next
```

The first time through this, #A is equal to 1, and a dialog box appears displaying this value with an OK button. Once the button is pressed, the loop starts over (because of the Next command), and #A is equal to 3, since the Step value is 2. This continues until #A is greater than 19 or, in this case, when it equals 21.

You can reverse the values with

```
For #A = 19 to 1 Step -2
```

Note that if you want to decrement, you must use a negative step value or your loop will last quite a long time. If you do not specify a Step value, it is assumed to be 1 if the start value is less than the end value, or -1 if the start value is greater than the stop value. Also, the equal sign, the word "to", and the word "Step" are optional. You could also code the previous example:

```
For #A, 19, 1, -2
```

You may nest For/Next loops. What that means is that a For/Next loop may be wholly contained within another For/Next loop. For instance:

```
For #A, 1, 10
For ![3], 1, 10
#C = #A * ![3]
Message   "#A = " & $#A & ", ![3] = " & $![3], $#C
Next
Next
```

You will see 100 dialog boxes appear one after another, displaying the progression of #A and ![3], as well as their product in the title bar.

There is a limit to how deep you can nest For/Next loops. It can be no more than 10 deep. This does not mean that you are limited to 10 For/Next loops; what it means is that you cannot have more than 10 For statements without a corresponding Next statement. Each Next statement subtracts one from the number of available For's. The following illustrates the progression:

```
For                 1 For
```

For	2 For's
For	3 For's
Next	2 For's
For	3 For's
For	4
Next	3
For	4
For	5
For	6
Next	5
Next	4
Next	3
For	4
Next	3
Next	2
Next	1
Next	0

Don't ever write your code so that it exceeds 10 in the For count column.

The Break statement tells EtCetera to prematurely exit from the current For/Next loop.   EtCetera will jump to the line following the next Next statement, forcing the termination of the current For/Next loop.   This can be used as follows:

```
FileFill "*.ETC", 100
For #A = 100 to Lines + 99
Ask $A, "This one?: " & $[#A]
If $A = "Yes" Then Break
Next
If #A > Lines+99 Then Goto NoFileChosen
...
```

When a For/Next loop completes, the variable used in the For statement will be either greater than or less than the end value, depending on whether or not the end value is greater than or less than the start value, respectively.   This can be checked at the end of a For loop to determine whether or not the loop was broken prematurely or if it completed.

See the descriptions of FileFill, Ask, If, and Goto for more information about the preceding example.

Match

Match is like a lesser FileFill command. It retrieves a single filename which matches the file mask provided. [By "mask" is meant a set of criteria - in this case, a filename or pathname, with or without wild card characters.] Match can be used repeatedly to retrieve additional files which match the file mask. The first time Match is called, it finds the first entry in the directory specified (or the current directory if no directory is specified) which matches the mask. Subsequent Match commands with the (exact) same mask locate the next entry which matches the mask. As soon as no additional matches are found (or if no match is found in the first place), a blank string is returned instead. Therefore, you can use this feature to find if a file exists by using the actual filename as the mask.

Once Match is used with a different mask, the process starts all over. So long as the same mask is used, Match will continue to retrieve files which match the mask (or until no more files match the mask). You cannot, therefore, use Match with two separate sets of masks simultaneously. You must use one mask first, and, once the desired file is located, move to the next mask - otherwise, you'll keep getting the same two files (the first ones to match each mask).

The format is

```
Match [string variable] [mask]
```

For example:

```
Match $A, "*.EXE"
```

Call

Call is used to load another EtCetera file and execute it from the beginning.   Its syntax is

Call [string]

where [string] is a filename or pathname to the EtCetera batch file you want to run.   EtCetera remembers all variables when a new file is called, but it does not remember the name of the previous file.   If you need to know the name of the calling file, save it in a variable.

Align

Align reorders all icons. If icons have been moved around the screen, Align repositions the icons neatly at the bottom of the display window. If the icons are already neatly arranged, this command does nothing. It takes no parameters.

Flash

Flash flashes the specified window a specified number of times. If no window is specified, it uses the active window. By "flash" is meant the title bar is toggled to the "activate" state and then "deactivate" state (or vice versa, if the active window is involved). Usually, unless both states are specifically set to the same color, the title bar for an active window and an inactive window are colored differently.

The format is:

Flash {string} [integer] {time(s)}

{string} is the unique title bar text indicating which window you want to flash. [See the description of Activate for information on creating unique title bar text.] [integer] is the number of times you want to flash the window. You can use a long variable for [integer], but it must not exceed the maximum value for an integer variable, 32767; It is very unlikely that you will want to flash a window more than 10 times, let alone 32767.

For example:

Flash 3
Flash "Notepad" 3 times

KeyOff, KeyOn, Toggle

The functions are used to set the states of the keys which have LEDs: NumLock, ScrollLock, and CapsLock. KeyOn turns on the LED; KeyOff turns off the LED; Toggle switches the state of the LED to its opposite. The format is

[command] [LED-key]

where [command] is the desired command, and [LED-key] is either a C, and N, or an S for CapsLock, NumLock, and ScrollLock, respectively. If desired, you can actually spell out the key name, but only the first letter is required.

You can only affect one key with a single command. If you want to turn on all three LEDs, use three separate KeyOn commands.

You can use the LEDs keyword to determine the state of these keys.

Special Keys

Below is a table which shows the special (non-text) keys supported by Send, SendKeys, and Type, and their representations. Note that the representation must be enclosed in curly braces (which must also be inside of quotation marks or in a string variable). This list does not include function keys or keypad keys, which are described in the description of Send, SendKeys, and Type. The representations are not case-sensitive.

Key	Representation(s)
Alt	Alt
BackSpace	Back, BackSpace, BkSp
CapsLock	Caps, CapsLock, CL
Ctrl	Ctrl, Control
Del	Del, Delete
Down Arrow	Down, Down Arrow
End	End
Enter	Enter, Return
Esc	Esc, Escape
Home	Home
Ins	Ins, Insert
Insert	Ins, Insert
Left Arrow	Left, Left Arrow
NumLock	Num, NumLock, NL
Page Down	Next, PageDown, PgDn
Page Up	PageUp, PgUp, Prior
Pause	Pause
Print Screen	Print, Print Screen, PrtSc, SnapShot
Return	Enter, Return
Right Arrow	Right, Right Arrow
Scroll Lock	Scroll, Scroll Lock, Scrl, SL
Shift	Shift
Tab	Tab
Up Arrow	Up, Up Arrow

Note that you can send Alt, Ctrl, and Shift as specific keystrokes, rather than solely as modifiers. Some Windows applications make use of these keys differently, especially those keys located on the right side of the keyboard. If you are want to indicate the versions of these keys on the right side of the keyboard, be sure to use the "/e" ("enhanced") modifier prior to the keytext. In other words,

```
SendKeys "/e{Control}"
```

would send the right Control key as a "character", without actually sending any other keys.

Keyword Reference

Keywords are words which are defined in EtCetera to have specific meanings. Keywords behave like numeric values or strings, and can generally be used in place of them or within equations (but never in place of a variable where one is required). The keyword descriptions are divided into groups based on the keywords' function.

Version keywords
Time/Date keywords
Screen Coordinate keywords
Window State keywords
Directory keywords
String Conversion keywords
System State keywords
Other keywords

Version keywords

DOSVer, ETCVer, WinVer

Version keywords return numeric values indicating version numbers. Each keyword returns the version number multiplied by 100, since EtCetera only supports integers (and cannot, therefore, use numbers with decimal points). There are three such keywords in EtCetera: ETCVer returns the version of EtCetera ; DOSVer returns the version of DOS (330 for DOS 3.30, 500 for DOS 5.00, etc.); WinVer returns the version of Windows (310 for Windows 3.10).

Time/Date keywords

Date, Day, Hour, Minutes, Month, Seconds, Time, Weekday, Year

There are nine time/date keywords in EtCetera.   Only two return strings; the rest are numeric.

The string keywords are Time and Date.   Time returns the time in military notation (i.e. 13:22:41).   Date returns the date in mm/dd/yy format (i.e. 10/24/92).

The numeric keywords are Day (returns the current date value), Month (the current month), Year (the current year), Hour (the current hour, in military 0-23 format), Minutes (the current minute value), Seconds (the current second value), and Weekday (the day of week).   Weekday returns a 0 for Sunday, a 1 for Monday, etc., through to 6 for Saturday.

Screen Coordinate keywords

Height, ScrH, ScrW, Width, XPos, YPos

Screen Coordinate keywords deal with the width, height, or position of items with respect to the video display. The video display is made up of individual points (called pixels). Different display types have different numbers of pixels both vertically and horizontally.

ScrW and ScrH return the width and height of the current video display in pixels, respectively. For a standard VGA-type display, this is 640 and 480.

Width and Height return the width and height of the active window in terms of pixels. A window which takes up one-fourth of the display area of a VGA display would have a width of 320 and a height of 240.

XPos and YPos return the x-coordinate and y-coordinate of the upper-left-most pixel of the active window. This is always with respect to the upper left corner of the video display, which is XPos = 0, YPos = 0 (written (0, 0)). If the upper left corner of an application's window is 30 pixels to the right of the upper left corner of the display and 70 pixels below it, XPos would return 30, and YPos would return 70. On a standard VGA display, the lower left corner of the display is (639, 479).

Window State keywords

IsMin, IsMax, IsOpen, IsWin

Window State keywords return values based on the current state of the active window. If a condition is true, these keywords return the value 1. If a condition is false, the return value is 0.

IsMax checks to see if the active window is in its maximized state. IsMin checks to see if the active window is an icon. IsWin checks to see if the active window is in its standard window state.

IsOpen checks to see if a window exists whose title bar text matches what you specify. For instance, you can check to see if Notepad is open and, if not, run it:

```
If IsOpen "Notepad" Then Goto :AlreadyOpen
Run "NOTEPAD"
:AlreadyOpen
...
```

See the descriptions of If/Then, Goto, and Run for more information.

Directory keywords

GetDirectory, SysDirectory, WinDirectory

Directory keywords return strings of various directories. GetDirectory gets the current DOS directory, including a disk drive identifier. For instance, if the current directory (i.e. if the program were "at the DOS prompt", what the prompt would show) is the root directory of your D drive, GetDirectory would return "D:\".

WinDirectory returns the complete pathname to the Windows directory, such as "C:\WINDOWS". SysDirectory returns the complete pathname to the Windows system directory, which contains much of Windows itself. This directory is almost always called SYSTEM, and it is always located in the Windows directory. So, for the WinDirectory example, SysDirectory would return "C:\WINDOWS\SYSTEM".

String Conversion keywords

ANSI, Left, LowerCase, Mid, Right, UpperCase

String Conversion keywords allow you to modify and manipulate strings. UpperCase and LowerCase take a parameter (a string) and return it converted into either uppercase or lowercase, respectively. Left and Right take two parameters, a string and a numeric value, and return as many characters from the "edge" of the string specified as is specified. Mid takes three parameters, a string and two integers. The first integer indicates a position from the left "edge" of the string, and the second integer indicates the number of characters to return from the position indicated with the first integer. The ANSI takes an integer parameter and returns a single character whose ANSI value is the integer specified.

To illustrate this, assume that the variable $A contains "John Doe". In this illustration, $B will be used to store the modification.

```
$B = UpperCase $A
```

In this case, $B would contain "JOHN DOE" after EtCetera executes the line above.

```
$B = LowerCase $A
```

$B would contain "john doe".

```
$B = Left $A, 3
```

$B would contain "Joh"

```
$B = Right $A, 4
```

$B would contain " Doe"

```
$B = Mid $A, 3, 4
```

$B would contain "hn D"

The ANSI keyword is a little bit different:

```
$B = Ansi(169)
```

$B would be the copyright symbol, ©. Refer to your Windows documentation for a list of supported ANSI characters and their associated values. You can also use ANSI to represent a carriage return, which is ANSI value 13.

Note that in none of the above cases is $A changed at all - and, so, the name "String Conversion keywords" is a little misleading. In EtCetera, the only variable which is ever changed in an equation is the one to the left of the equal sign.

System State keywords

ActiveChild, ActiveWindow, CPU, ErrVal, Files, LEDs, Lines, Mode, NumApps

System state keywords return information about the behavior of the system.

ActiveChild returns the complete title bar text of the active child window. A child window is a document window (such as a single spreadsheet). ActiveWindow returns the complete title bar text of the active window. In both cases, by "active" is meant the window which receives any keystrokes you type.

CPU returns a string which indicates what kind of microprocessor is in the computer: "8086", "80186", "80286", "80386", "80486". "8086" could also be an 8088, and "80186" could be an 80188.

ErrVal returns the number of the most recent error. The value remains the same until another error is encountered.

LEDs returns a three-character string which indicates the state of the LED keys on the keyboard. If the Caps Lock LED is on, the first character will be a "C". If the Num Lock LED is on, the second character will be an "N". If the Scroll Lock is on, the third character will be an "S". If any of these LEDs are off, its corresponding position will be a space instead of a letter.

Mode returns a string which indicates the current operating mode of Windows: "Real mode", "Standard mode", or "386 Enhanced mode".

NumApps returns the number of applications which are currently executing. It does not include EtCetera.

Files returns the number of files on which the most recent command which operates on files (such as Copy) affected. If Copy copies three files, Files returns 3.

Lines returns the number of entries in the string array structure which are modified by a command which uses the string array structure. If FileFill copies six filenames into the string array structure, Lines returns 6.

Other keywords

Clipboard, GetAttr, Length, Parameters, TempFileName

These keywords do not fall into any other category.

Clipboard returns the text in the clipboard. If there is more than 255 characters of text in the clipboard, EtCetera truncates the text.

GetAttr gets the file attributes of the file you specify. This attributes are defined according to the file's directory entry. Each attribute has a specific value (which is a power of 2, such as 1, 2, 4, 8, 16...).

Read-Only:	1
Hidden:	2
System:	4
Archive:	32

The following code illustrates how to show how to determine which attributes apply to a file:

```
#A = GetAttr "WIN.INI"
If #A >= 32 Then Message "Archive"
If #A >= 32 Then #A = #A - 32
If #A >= 4 Then Message "System"
If #A >= 4 Then #A = #A - 4
If #A >= 2 Then Message "Hidden"
If #A >= 2 Then #A = #A - 2
If #A >= 1 Then Message "Read-Only"
```

GetAttr takes a string for a parameter, which should be a valid, existing file.

Length takes a string for a parameter and returns the number of characters in the string. For example,

```
#L = Length "This is a test."
```

Parameters returns a string which is the command line parameters for EtCetera. For instance, if you run EtCetera with the command line (per the File Properties option in Program Manager)

```
ETCETERA GETFILES.ETC /COM1 /9600 /8N1
```

then Parameters returns "GETFILES.ETC /COM1 /9600 /8N1".

TempFileName retrieves a filename which is guaranteed to be unique within the system. The string returned is a complete pathname to a file which does not exist (already). This file will be in the directory indicated by the TEMP environment variable, which should be set in your AUTOEXEC.BAT file. Refer to your Windows documentation for more information. If the TEMP environment variable is not set, then the file will be in the Windows directory or the root directory of your boot drive.

You can use TempFileName to find a filename which you can use which will not accidently overwrite some other file. Do not use TempFileName if you do not plan to use the file right away, as Windows returns the filename, and it could just as easily give it someone else if the file is found to be nonexistent when another program requests a unique filename.

Data Formats and Variables

EtCetera supports two data formats. The first is a numeric format. Numbers can be positive or negative, but they must be integers (i.e. they cannot have a decimal point). Do not use commas when writing numbers. To write ten thousand, use 10000, but not 10,000. Depending on the context, EtCetera can use numbers between negative two billion and positive two billion. [Technically, between -2147483648 and 2147483647. This is due to the way computers store numbers internally.]

The other data format is textual data, called strings (short for "character strings", or "strings of characters"). A string can have a maximum length of 255 characters. Strings can also be empty.

Variables are storage locations in memory which can vary - hence the name. Variables are used to store data temporarily while a program operates on the data. EtCetera supports four kinds of variables. Two of these variable types are numeric. The third type is for strings. The fourth is a special type called a handle.

Numeric variables

EtCetera supports two distinct numeric variable types. The first is a standard integer, and the second is a "long integer" or a long.

Integers may range from the value -32768 to the value 32767. [Again, this is due to the way computers store numbers internally.] If an equation or an assignment exceeds either of these values, EtCetera will generate an error. Long integers may range from -2147483648 to 2147483647, and if these values are exceeded, then the long will "roll-over". For example, -2147483649 becomes 2147483647.

An integer variable is specified with a pound symbol (#) and a single alphabetic character (A-Z). To assign a value to an integer variable, use an =, as

```
#A=32
```

Complex equations are supported, such as

```
#A = (12+#B) * #A
```

When EtCetera evaluates an equation, the order in which the equation is evaluated follows the standard mathematical evaluation criteria. Portions of an equation within parentheses are evaluated first. Then, multiplication (*), division (/), and modulus (%) are performed. (Modulus returns the remainder of a division operation.) Then, addition (+) and subtraction (-) are performed. Use parentheses to alter the order of operation. Parentheses may be embedded within parentheses.

In addition to the variables #A through #Z, EtCetera supports an integer array structure. This is a group of 1000 separate integers which can be accessed individually. To access one of these items, use

```
#[value]
```

where value is the entry number you want to use. Valid entry numbers are 0 through 999. For example,

```
#[100] = 13
#[#A] = 24
#[#[#A]] = 19
```

are all valid. The third entry shows a particularly powerful use of arrays. Entries can be nested. The innermost item is evaluated first, followed by that array item, followed by the outermost array item. Assuming #A is equal to 24, and #[24] is equal to 13, this command would set #[13] equal to 19.

Long variables are denoted with an exclamation point, as in !A or ![234]. Like integers, longs can be used in equations, and longs have their own array structure. You can mix and match integers with longs in equations, but if you assign a value to an integer which exceeds the limits of an integer variable, an error will occur.

Besides literal numerals and variables, there are keywords which return numeric values. These special words behave like numbers and can be used in place of them where a number (but not a numeric variable) is required.

String variables

A string is a series of alphanumeric and other characters. Strings are limited to 255 characters. A string variable is indicated with a dollar sign ($) followed by a single alphabetic character (A-Z). To assign a string to a string variable, use an =, as

```
$A = "Yes!"
$B = $A
```

Literal strings are designated by surrounding the text in quotation marks. To indicate a quotation mark should be a part of the text, place two quotation marks together: "Say ""what""" will be understood by EtCetera to be

```
Say "what"
```

EtCetera supports complex string functions or concatenation. Concatenation is performed by listing two or more strings, variables, or string keywords, and separating them with an ampersand (&). For example:

```
$A = "This "
$B = "is a "
$C = "test."
$D = $A & $B & $C
```

$D is "This is a test." after EtCetera executes these four lines.

If you want to convert an integer variable into text, preface the integer variable with a $, as in

```
Message $#C
```

which would display the value of #C, as text, in a dialog box. (See the description of the Message command for more information.)

In addition to the variables $A through $Z, EtCetera supports a string array structure. This is a group of 1000 separate strings which can be accessed individually. To access one of these items, use

```
$[value]
```

where value is the entry number you want to use. For example,

```
$[13] = "This is a test."
```

assigns "This is a test." to entry number 13. The valid numbers are 0 through 999. See the description of the integer array structure in Integers, above, for a more detailed description of the use of arrays.

The string array structure differs from the numeric array structure in that the string array structure plays an integral part (pun intended) of several of EtCetera's commands. See the descriptions of FileFill, GetClipboard, ReadFrom, SetClipboard, TitleFill, WriteTo.

Handles

A handle is an indirect reference for something. Much of what Windows does is based on handles. Handles are really just numbers, and these numbers have meaning to Windows internally, but the numbers are used by programmers to tell Windows which of something it should do something to. Confused?

It's like renting a Post Office Box and telling everyone to send your mail to the P.O. Box. Doing so gives you the freedom to change your address or go live in the park, but so long as you can get to the Post Office yourself, you can get your mail.

Likewise, using handles allows Windows to tell you how to get to some item (such as a window) without having to tell you a whole lot about the way Windows treats it, or what it takes to access it or manipulate it.

Handles are represented in EtCetera with an @ symbol, such as @A or @[26]. There is a handle array structure, just as there are string, integer, and long array structures. Handles cannot be used in any form of equation, as they should not be manipulated. They can be used by commands and are returned by certain commands, but something of the form

```
@A =
```

will automatically generate an error. After all, the Postal Service tells you which P.O. Box you get, or which Zip Code is assigned to which area, but not the other way around.

Introduction

EtCetera is a batch language interpreter for the Microsoft Windows environment. A batch language is a simple language which can be used to simplify or automate repetitive, redundant, or boring tasks. DOS has a batch language "built in", but Windows does not have one. [Please do not inform Microsoft of this oversight.] EtCetera adds this much-needed functionality to Windows. Using EtCetera, you can reduce a series of tasks to a double-click.

EtCetera is simple to program. EtCetera has an editor mode which allows you to enter commands into EtCetera and test them right away. To enter this mode, simply double-click the EtCetera icon in Program Manager. [If you use a different shell program for Windows, this may behave differently.] When you run EtCetera without any command line parameters, EtCetera enters editor mode.

If, instead, you specify a filename on the EtCetera command line [EtCetera files end in the extension ETC by default], EtCetera loads the file you indicate and attempts to run the program. Once the program is done, EtCetera unloads itself from memory.

EtCetera batch files are simple text files. You can create them in editor mode, or you can type them in in Notepad. You can also create them in a word processor, but you must be certain to save them in text-only format.

Each line in the file is a single command and is terminated with a carriage return (press Enter). Only one command can be placed on a single line [see the description of the If/Then command for the only exception].

There are ten items of importance to EtCetera: Commands, Keywords, Mathematical Operators, Logical Operators, Modifiers, Variables, Numerals, Strings, Labels, and Comments. The paragraphs below describe these items briefly. The remainder of this help file more fully explains each.

Commands tell EtCetera to do something, such as run another program or simulate keystrokes. Commands frequently take parameters, which are other items which define the behavior of the command. This can be the name of the program to run or the keys to simulate.

Some commands have special words which are unique to the command (or group of commands), called Modifiers (or Options). The descriptions of each command list any Modifiers for a particular command.

Numerals are simply that - numbers - like 10000 or -18. In EtCetera, integer numerals are supported. You cannot use a decimal point with a number (a real number). Batch languages generally deal with concrete, whole things, not parts of things.

Strings (short for "character strings") are series of text. A string can contain up to 255 characters or as few as 0 characters (a blank string). Strings are surrounded by quotation marks.

Variables are storage areas in the computer's memory. Variables can be used to store numeric values or strings.

Keywords are specially reserved words which can be used in place of numerals or strings. Keywords behave as if they were the values they represent. For instance, the keyword DOSVer represents the version of DOS running on the system.

Mathematical Operators are used to create equations. These include +, -, and =. Strings can also be used in equations.

Logical Operators are used to make comparisons (and, at this point, are exclusive to the If/Then command). The equal sign = is a logical operator (and behaves differently than its mathematical twin when used this way). Other logical operators include < and >.

Labels are used to mark a specific line in a program. (See the Goto command.) A label is always preceded by a colon and consists of textual characters. It is terminated with a carriage return or a space. For example,

```
:ThisIsALabel
```

Comments can be used to describe sections of your program. A comment line is begun with a semi-colon, as

```
; This is a remark.
```

Everything on such a line is ignored by EtCetera. Using comments can make it easier to understand the behavior of your batch file, particularly if someone else is going to use it.

In general, you should separate commands, modifiers, equations, and some logical operators (those which are represented as text rather than symbols, such as < and =) with at least one space or comma. EtCetera uses these symbols to mark the end of one and the beginning of another. Spaces are not required within equations, so long as an operator of some sort falls between the various items in the equation. In general, adding spaces and commas to your programs will make it easier to read the program later. The sample code shown in the rest of this help file should help you understand the nuances of programming EtCetera.

You can also review the sample code contained in the demo files, included with EtCetera, called ETC_DEMO.ETC, NOTE_TUT.ETC, NOTESTUT.ETC, and PLAYWAVE.ETC. ETC_DEMO.ETC is a general sample file. NOTE_TUT.ETC is a mini-tutorial which shows the user how to choose items from the menu bar. NOTESTUT.ETC is the same file, but it includes .WAV (multimedia sound, or wave) files, and a voice actually speaks the text of the tutorial as it appears on the screen. You must have a sound card and Windows 3.10 or Windows 3.00 with the Multimedia Extensions in order to hear the sound. PLAYWAVE.ETC displays the names of all WAV files in the Windows directory as it plays them in succession.

Throughout the descriptions of commands and keywords, the formats of these commands show which parameters are required and which kinds of items are optional. Required items are surrounded by square brackets [], and optional items are surrounded by curly braces {}. In the sample code in this help file, a set of ellipses (...) are used to indicate that some other code would follow, but its presence is not required to illustrate the example.

What is ETCETERA.386?

ETCETERA.386 is a module which adds some functionality to EtCetera 2.10. It uses special functionality available only to 386 and higher processors, and only to .386 files under Windows. These files are commonly called Virtual Device Drivers, or VxD's, so named because these files are generally used to permit sharing of various devices installed in your PC, such as the hard drive, a sound card, a network card, your printer ports, your modem ports, etc. (By "virtual" is meant that it makes each program that runs think that it has its own "copy" of the device, or a "virtual" device, or, if the device cannot be shared, it makes the program think that the device does not exist.)

ETCETERA.386 is not technically a device driver, since it does not actually control access to any particular device. It does, however, allow EtCetera to use the full power of 386 and later processors, as well as access Windows in ways not otherwise possible. It is the use of this file alone that makes the 386-only commands supported in EtCetera available.

To install ETCETERA.386 (and make its functionality available), use Notepad or SysEdit (System Editor) to open your SYSTEM.INI file. It is located in your Windows directory. Locate the section which begins with

```
[386enh]
```

and insert a line immediately following it which reads

```
device=C:\WINDOWS\ETCETERA.386
```

This assumes that you copied the EtCetera program files into your Windows directory. Modify the path name accordingly.

Once the line is typed in correctly, save the SYSTEM.INI file, and then restart Windows. If ETCETERA.386 is properly installed, and as long as you are running in 386 enhanced mode, you will see the words

Running in Enhanced Mode

in the banner display. If you are not running in 386 enhanced mode, or if ETCETERA.386 is not properly installed, then

Running in Standard Mode

will be displayed. The only difference between Enhanced Mode and Standard Mode is that you cannot use the 386-only commands in Standard Mode.

Menu Commands

EtCetera will run in one of two modes. If EtCetera is run without including a batch file name on the command line, EtCetera will come up in editor mode. In editor mode, a window will appear in which you can type batch files and test them. This editor is similar to the Windows Notepad program. If you are familiar with standard Windows menus, then EtCetera's menus should be simple to use.

The File menu has six commands. The New command will erase whatever is in memory and let you start over from scratch. If something is currently in memory, you will be asked whether or not you want to save the file first. The Open command can be used to load a batch file from disk. Again, if something is currently in memory, you will be given the option to save whatever it is prior to loading a new batch file.

The Run command allows you to test batch files by beginning execution of the file. If you are done programming a file or need to go on to something else, you can use the Save and Save As commands to save whatever is in memory. The Save command will save the file using whatever the current file name is. This file name appears in the title bar at the top of the EtCetera window after the hyphen, like

EtCetera - AUTOMATE.ETC

This name is either the name of the file opened or the most recent name used to save the contents of memory. If, instead, the title bar reads

EtCetera - (untitled)

then no name has yet been assigned to this file, so EtCetera will prompt you for one.

If you want to save the contents of memory with a file name other than the one in the title bar, the Save As command will prompt you for a file name just as if the title bar read (untitled).

When you are finished using editor mode, the Exit command will close the EtCetera window and remove it from memory.

Under the Edit menu are four commands: Cut, Copy, Paste, and Goto. The Cut command will remove any highlighted text from the editor window and place it into the Windows clipboard. The Copy command will place a copy of any highlighted text from the editor window into the Windows clipboard. In order for these commands to be available, there must be some text highlighted. To highlight text using a mouse, move the mouse to the beginning position of the text you want to cut or copy, depress the left mouse button, drag the mouse in the direction of the end position of the text you want to cut or copy, and then release the left mouse button once the mouse pointer arrives at the end position of the text to cut or copy. To highlight text using the keyboard, move the caret (the blinking vertical bar) to the beginning position of the text you want to cut or copy using the arrow keys, then, hold down either shift key and use the arrow keys to move the caret to the end position of the text you want to cut or copy - then release the shift key.

The Paste command will take any text currently in the Windows clipboard and either insert it where the caret is located or, if some text is highlighted, will replace the highlighted text with the text in the clipboard, if available. If the clipboard does not contain any text data, the Paste command will be unavailable.

The Goto command will move the caret to the line whose number you specify. If an error occurs in your program, EtCetera will report the line number where the error occurred. You can use the Edit Goto command to quickly locate the offensive line.

The Insert menu has two items - Commands, and Keywords. Next to each of these is a small arrow, indicating that these are cascading menus - menu items which have menus. If you select one of these items, a second menu will appear. The Commands menu shows a list of all available commands in EtCetera. You can choose one of these, and EtCetera will insert the command with a description of required or optional parameters. The Keywords menu behaves similarly.

The Help menu has three items. The first is Index, which you use to access this help file. The second is Help On Help, which runs the Windows help program and loads a file which describes how to use help. If you have deleted this file, or if Windows cannot locate the file, then the Windows help program will display an error message.

The last item is the About EtCetera command. Choosing this item will display a dialog box with copyright information.

Programming EtCetera - Tips

Programming EtCetera is similar to programming in BASIC or in the MS-DOS batch language. There are words which tell EtCetera to perform various tasks. These words are called commands.

EtCetera reads a file, one line at a time, and interprets each word, letter, number or symbol into something meaningful. If it cannot interpret the information, or if EtCetera is unable to utilize the information at the time it interprets it, a dialog box will appear indicating the nature of the error.

Commands and Modifiers

The following line is a sample EtCetera line:

```
Message "This is a test."
```

This command, Message, tells EtCetera to display a small dialog box containing some text and an OK button. The text it will display is represented by "This is a test." The sentence

This is a test.

appears in the dialog box above the OK button.

The portion of the command "This is a test." is called a parameter. What is meant by "parameter" is something which modifies the behavior of the command. Effectively, then, a parameter is practically anything appearing on a line, following a command, which is not the command itself. When a command is capable of accepting one or more parameters, it is said to "take" a parameter or parameters.

The Message command, listed above, is an example of a command which can take a variable number of parameters. Many of EtCetera's commands can take variable numbers of parameters. Some take a fixed number of parameters. Others take no parameters at all.

Additional examples of the Message command are:

```
Message "Are you ready to proceed?", "LogOn"
Message "Unable to access file.", STOP
```

In each of these cases, Message takes two parameters, although clearly the two types of parameters taken differ. In the first case, the second parameter, "LogOn", tells EtCetera to place the text

LogOn

in the title bar of the dialog box. In the second case, the second parameter tells EtCetera to place a stop-sign icon in the dialog box to the left of the text of the dialog box.

The Message command can take up to five separate parameters. You can view the description of Message in the Command Reference portion of this help file.

In the above examples, the text to be placed in dialog boxes was surrounded with quotation marks. Quotation marks tell EtCetera to take the text contained inside as a single unit, known as a string. Without the quotation marks, EtCetera will attempt to interpret the word as something it understands. Since, for example, the word Unable means nothing to EtCetera, EtCetera would display an error message inside a dialog box indicating that the line in question has one or more invalid parameters if the quotation marks were not used.

The word STOP, however, is understood by EtCetera within the context of the Message command. STOP tells EtCetera to place a stop-sign icon in the dialog box. There are other commands where the word STOP is similarly used. The word STOP itself is also a command when it begins a line in the program file. Therefore, the context (other words, or lack thereof, both before and after the word in question, which help define the word's meaning) is important. In one case, STOP means one thing; in another case, it means something completely different. Interestingly enough, the command Stop can take the word STOP as a parameter.

As a command, Stop tells EtCetera to stop running the program file currently in memory and also to display a dialog box much like Message - therefore, the modifier STOP can be used with the command Stop to display a stop-sign icon in the dialog box which Stop displays.

When STOP is used to mean "display a stop-sign icon in the dialog box", it is called a modifier. Throughout this text, modifiers will be differentiated from commands in that commands will be displayed in mixed case (i.e. Stop), but modifiers will be displayed in all capitals (i.e. STOP). This should help to eliminate any confusion. To EtCetera, however, it does not matter whether or not you use capital letters or lowercase letters to indicate whether or not Stop is a command or a modifier. The position of the word on the line with respect to other words tells EtCetera which one it means.

In the Message commands above, notice the spacing and punctuation of the commands. There are spaces between words, and sometimes commas. With EtCetera, you should always separate the command from the rest of the line with a space, much as you would in English. Parameters may be separated with spaces and/or commas, per your preference. When you have several parameters on a line, some of which are strings containing many words, reading the line and understanding it quickly can be difficult without commas, which you are probably already trained to observe. Compare the following lines:

```
Message "This is a test" "This is the title bar" STOP
```

```
Message "This is a test", "This is the title bar", STOP
```

Most people will agree that it is easier to distinguish where one parameter ends and another begins in the second line than in the first line. If you do not agree, then as long as you do not intend to share your programs with anyone else, you can do what you like. It does not matter to EtCetera whether or not there are commas.

Numerals, Coordinates, Expressions

Consider the following line:

```
Move 100, 100
```

The Move command tells EtCetera to relocate the active window to a specific position on the display screen. Your screen is divided into many small square dots called "pixels". A pixel is the smallest indivisible light source available on your screen, akin to a cell in your body or an atom in a molecule. Depending on the monitor and circuitry used to operate the monitor, your screen may have over a million pixels.

Whenever referring to pixels, each pixel has a specific "address", called its coordinates. Coordinates describe how far to the right and how far down a specific pixel is from the upper left corner of the video display. The pixel which is the extreme upper left corner of the display has the coordinates (0, 0). For every pixel to the right, one is added to the first number, so that the next pixel to the right of the upper left corner has the coordinates (1, 0), and the one following that is (2, 0), etc. How far down a pixel is relative to the upper left corner of the display is indicated by the second number.

If you have a VGA display, then your display is made up of a field of pixels which is 640 pixels wide and 480 pixels deep. This means that the pixel which is the lower right corner of the display has the coordinates (639, 479).

Whenever you use coordinates with EtCetera, the coordinates always follow that behavior - the coordinates are given with respect to the upper left corner or the next larger item. In other words, when you are referring to a window, the coordinates are relative to the upper left corner of the video display. When you are referring to something within a window, such as a picture or some text, the coordinates are relative to the upper left corner of the window containing (or which will contain) the item in question. In both cases, the coordinates describe the location of the upper left corner of the item in question.

Since the Move command affects a window, the parameters 100 and 100 tell EtCetera to move the active window so that its upper left corner is 100 pixels to the right of the upper left corner of the video display and 100 pixels down from the upper left corner of the video display. Windows, by default, uses the upper left corner of any object as its point of reference. EtCetera adheres to that behavior.

As suggested by their use above, EtCetera can interpret numerals. EtCetera is limited to numbers which do not have a fractional portion (such as 3.45, or 2.11111111...). Since a batch language is designed to handle real-world items, and since you will seldom desire to perform one-third of a job, this limitation will not generally be a problem.

EtCetera also supports expressions. EtCetera can evaluate various items and treat them as a single unit, such as

```
Move 25*4, 33*3+1
```

which becomes, effectively,

```
Move 100, 100
```

The use of the commas are optional. Note in this case, however, that the commas separate individual parameters. Do not use commas within numbers, such as 100,000 for "one-hundred thousand". EtCetera views 100,000 as "one hundred, zero".

"An expression" is meant here to be "an equation without an equal sign." Technically, any parameter could be called an expression, but expression is used here to mean a parameter made up of at least two distinct parts. Generally, an expression will have some form of mathematical or other symbol to connect the two or more items which make up the parameter, since spacing without one of these symbols means "next parameter".

Obviously, it is much simpler to evaluate the expression and use the genuine value rather than force EtCetera to do the work of evaluating the expression. The reason for using such an expression is when variables come into play.

Variables and Equations

Here is an equation which uses variables:

```
#A = 34 + #C
```

A variable is a storage area in memory which is capable of being changed by the programmer (or, indirectly, the user). In this case, #A and #C are variables which store integer numbers. Integers can be either positive or negative. Any variable which is begun with the # symbol is an integer variable. Integer variables can range between -32768 and 32767. This is due to the way computers store numbers and has nothing to do with a personal preference for those numbers.

What the above equation means to EtCetera is "take the value of 34, add it to the value of #C, and store the sum in #A." In other words, EtCetera looks to the right of the equal sign before it looks at the left. Once it is done evaluating the expression to the right, it completes the equation by placing the value from the right into the variable on the left.

Therefore, the left side of all equations must have a variable and only a single variable. The following does not work:

#V + #C = 34 - #D

The reason for this is that an "equation" to EtCetera is actually an "assignment". The item on the left is assigned the value evaluated from the right. This is not algebra. In a true, valid equation, a fact is being stated. In an assignment, a fact is being made: the item on the left is being made equal to the value on the right.

Therefore, the following, which would totally collapse the universe if it were true as an equation, is not a problem for EtCetera, for which it is an assignment:

#A = #A + 100

If #A contains the value 10, then after this line is interpreted by EtCetera, #A contains 110, which is the previous value of #A added to the value 100.

You will use variables for keeping track of how many times something is done, or for storing the current date, or any number of other numeral-based data which are important to you. Equations can be made as complex as you have space for. The description of data types and variables contains more information (available from the first Index screen).

General Information

A line in EtCetera can contain a maximum of 255 characters. Any characters on a line which exceed the 255-character limit are ignored. A line is terminated by pressing the Enter (or Return) key, just like you would for a word processor or Notepad.

Files saved by EtCetera will have the extension ETC. (For information on filenames and extensions, refer to your DOS documentation.) You can override this if you want - the extension is not important. Since EtCetera will look for filenames whose extensions are ETC by default, however, using a different extension will require an additional step on your part to open the file for editing.

Errors Reference

When EtCetera does not understand something in your program file, it will display an error dialog box. The dialog box will explain the nature of the error, as well as tell you the line number on which it occurs. The error messages are descriptive if you already understand what they are for, but when you encounter them for the first time, you may not understand them - especially if the error message is unique to a specific command. The error messages you will find in EtCetera are discussed below.

The format of all error messages is

XXXXX on line Y. Continue with next line?

XXXXX is the description of the error, and Y is the number of the line on which the error occurred. You can use the Edit Goto command to move the caret (the blinking vertical bar) to the line reported by the error message.

There is one error which could possibly occur, but, if it does, then you have encountered an extreme situation. The error message is:

An unknown and/or unrecoverable error has occurred on line X. Terminating current batch job. Please report the problem to Thetaware.

where X is the line number where the error occurs. The title bar of the error dialog box will contain:

EtCetera - Fatal Error #ABC

ABC will be some number. If this error occurs, one of two situations has arisen. The first possibility is that Windows has gone completely haywire and should be shut down and restarted. This first possibility could be caused by a virus, a bug in Windows, or some other application which has done something it is not supposed to do. The second possibility is that EtCetera's pieces are not communicating properly with each other. This is a result of poor programming on our part, and, of course, would be extremely rare.

If this ever occurs, please attempt to duplicate the problem. Please note the command which causes the problem, including the complete text of the line which causes the command, and the fatal error number displayed in the title bar, and forward the information to Thetaware. Registered users reporting such an error will receive a free update to the version of EtCetera in which this error is corrected if we caused the error.

You can use the error handling commands (Continue, ErrOff, Handle, OnErr, Retry) to prevent the default behavior for many of the errors. Included with each error is its error number which is contained within the ErrVal keyword whenever an error is detected. If no error number is given, then the error cannot be trapped.

Below is an alphabetical listing of the errors you can encounter in EtCetera, preceded by thier corresponding error codes:

520	A DOS error has occurred
4	An empty string was used for a parameter which requires a non-empty string
531	A network error has occurred
6	Command failed
100	Data string storage allocation error
530	DDE is not operating properly
514	Divide by zero error
3	Expected to find an equal sign for the equation begun
521	File called is too large
526	Invalid key option specified

2	Invalid parameter(s)
522	Maximum number of nested FORs (10) has been exceeded
519	Missing ']' on array
524	Missing NEXT for BREAK
1	Missing parameter(s)
525	No FOR loop in progress for BREAK
523	No FOR loop in progress for NEXT
5	No such label error
102	No such menu exists as specified
105	No window exists with the handle specified
101	No window exists with the title specified
513	Not a valid equation specified
532	Not an MDI application error
527	Not enough memory to allocate buffer
517	Out of memory error
516	Overflow error
512	Parameter error
515	Remainder by zero error
528	The CreateWindow command was unable to create a new window
529	The file specified is not a valid bitmap file
other	Undefined error
-1	Unknown command
103	You cannot use the command specified

A DOS error has occurred

Error number 520.

This error means that an attempt to do something to a file has failed. This can occur if you attempt to Execute a program, but the name of the program listed in the EtCetera file is misspelled. This error always occurs if a file does not exist. This error can be common with the commands Copy, CD, Rename, Run, and the like.

This can also occur if a directory does not exist, or an attempt is made to create a directory with a name which already exists as a file. In other words, this error occurs when anything occurs which DOS cannot handle.

An empty string was used for a parameter which requires a non-empty string

Error number 4.

This error is pretty self-explanatory. The following command will always generate this error:

```
Execute ""
```

Execute must have a filename to execute, and a blank string can never be a filename. Likewise,

```
Hide ""
```

will generate an error, because you must specify the text of a window's title bar in order to use Hide in this way.

A network error has occurred

Error number 531.

This error occurs only when using commands which in some way relate to the network (LAN). This could mean that an attempt to connect or disconnect your LAN printer, for instnace, failed for some reason. This error also occurs if you attempt to use a network command but do not have any network software installed.

Command failed

Error number 6.

This error means that, for one reason or another, the command on the line in question was unable to do what it was supposed to do.

Data string storage allocation error

Error number 100.

This error means that no space remains for creating a string, whether as a variable or internally for EtCetera itself in evaluating your program. Windows is limited to slightly over 16000 such strings (even if you have enough memory to create 16000000 of them). It is therefore unlikely that you will run out of space, but if many programs are using Windows' internal string storage area, then this space could become exhausted.

In general, this error is very unlikely.

DDE is not operating properly

Error number 530.

This error occurs when using any of the DDE commands and any failure occurs. This failure can be related to attempting to establish the DDE link, attempting to transmit or receive data, or a low memory situation, among others.

Divide by zero error

Error number 514.

This error occurs when you attempt a division operation which contains a zero as the denominator. A number cannot be divided by 0. If you need an explanation for this, please contact us and we will be happy to provide it.

Expected to find an equal sign for the equation begun

Error number 3.

This error means that a line was started with a variable rather than a command, but no equal sign was included on the line immediately following the variable. Remember that only one variable can be to the left of the equal sign, and, if you start a line with a variable, you must follow the variable immediately with an equal sign (spaces are allowed).

File called is too large

Error number 521.

This means that you attempted to use a CALL command to call a program file which exceeds 32K in size. EtCetera's files are limited to 32K in size.   You should break the file into smaller pieces.

Invalid key option specified

Error number 526.

This error occurs when you use a command which requires an LED-key specification, such as KeyOn, and use an LED-key specification which EtCetera does not understand. Keys with corresponding LEDs are NumLock, CapsLock, and ScrollLock. Their representations are N, C, and S, respectively. Case is not important. Any other letter used will generate this error.

Invalid parameter(s)

Error number 2.

This error means that you have used parameters which do not make sense for the command with which they are used. For example, the line

```
Activate 400
```

does not make sense, since Activate requires either title bar text or a window handle.

Maximum number of nested FORs (10) has been exceeded

Error number 522.

Read the description of Break, Next, For for information on FOR. EtCetera can have a maximum of 10 FOR loops active at one time. If an 11th active FOR loop is attempted, this error is generated. It is extremely unlikely that you would ever need 5 active FOR loops, let alone 10, so this error is equally extremely unlikely.

Missing ']' on array

Error number 519.

This error occurs if the end of a line is reached and there are one or more left square brackets (used for the various array structures) do not have corresponding right square brackets. This error will not always occur if square brackets are missing, but if this error occurs, then there is at least one unpaired left square bracket.

Missing NEXT for BREAK

Error number 524.

This error occurs when a BREAK statement is encountered in a FOR loop, but there is no NEXT statement following the BREAK statement.   Verify that all of your FOR statements have corresponding NEXTs.

Missing parameter(s)

Error number 1.

This error occurs when one or more required parameters for a command are not present. For example,

```
Move 100
```

generates this error, since all pixels on your video display requires two numbers to uniquely identify them. In other words, the second number is missing from the command above.

No FOR loop in progress for BREAK

Error number 525.

This error occurs only if a BREAK statement is encountered but there is no active FOR loop. This could be that you have no FOR loop in your program, or any FOR loops you have either have not begun or have already completed.

No FOR loop in progress for NEXT

Error number 523.

This error occurs if a NEXT command is encountered but there is no active FOR loop with which to match the NEXT command. This error usually occurs because an extra NEXT statement was accidentally placed into the program file.

No such label error

Error number 5.

This error occurs with the Goto and OnErr commands when the label specified following the command does not exist. Remember that labels are case-sensitive, and do not include a colon when naming a label on the Goto command line. [Only the label itself requires the colon.]

No such menu exists as specified

Error number 102.

When using the ChooseMenu command, if you specify more numbers than corresponding menus, you will get this error.   For instance,

```
ChooseMenu 1, 3, 5
```

is not valid if item 3 on menu 1 is not a cascading menu.

No window exists with the handle specified

Error number 105.

This error occurs when you specify a handle for use in a command when the handle was not retrieved using a different command.   For example,

```
PaintWindow @A
```

will fail if you have not used a command such as

```
CreateWindow @A
```

or

```
GetDesktop @A
```

to make @A a real handle.

No window exists with the title specified

Error number 101.

This error occurs when you specify title bar text but no window exists whose title bar has that text. For instance, if Notepad is not running, then

```
Activate "Notepad"
```

will return this error. Remember that title bar text is case-sensitive.

Not a valid equation specified

Error number 513.

This error means that the text of an equation cannot be interpreted. Usually, this means a typo.

Not an MDI application error

Error number 532.

This error occurs when you attempt to use any of the MDI commands when the active application is not a true MDI application.   [Refer to your Windows documentation for more information on MDI - the Multiple Document Interface.]   In order to be a true MDI application, the application must conform to certain specifications set forth by Microsoft.   Windows' Program Manager is a true MDI application.   Many of Microsoft's own applications (all of which have the ability to work with mutliple documents) do not follow Microsoft's own conventions for using MDI, however, so the MDI commands will not work, and you will receive this error.

Not enough memory to allocate buffer

Error number 527.

This error means that EtCetera asked Windows to set aside some memory for EtCetera's use, but Windows reported that it was unable to do so. For instance, the Copy command requires the use of a 32K buffer. If Windows does not have enough memory to set aside a 32K buffer, then you will get this error. The solution is to get more RAM or to close one or more running applications.

Out of memory error

Error number 517.

This error means that no memory is available to do something which you have requested from EtCetera. Try closing some unneeded application(s) and retry the program.

Overflow error

Error number 516.

An Overflow error occurs when you attempt to store a number in a variable which is larger than the variable can hold.   For example, if !A is equal to 1000900, then

```
#[24] = !A
```

since #[24] is a standard integer and cannot hold a value greater than 32767 or less than -32768.

Parameter error

Error number 512.

This error indicates that there is some expression which EtCetera cannot evaluate properly.   Usually, this is a typo.

Remainder by zero error

Error number 515.

This error occurs when you attempt to use the modulus operator (%) with a 0 numerator. Since modulus is division, you have the same limitations imposed which you would if you were using division. See the Division by zero error description.

The CreateWindow command was unable to create a new window

Error number 528.

For some reason, Windows refused to allow EtCetera to create a window for you.   You may have too many applications running.

The file specified is not a valid bitmap file

Error number 529.

The filename you entered for the DisplayBitmap command is not a valid bitmap file. Make sure you are using the right filename and, if so, attempt to open the bitmap with the PaintBrush program. If PaintBrush is able to read the file, try re-saving it and see what happens. If the problem persists, please report it to Thetaware.

Undefined error

This error has no defined value. Any error code returned which is not otherwise defined will generate this dialog box.

This means that an error occurred which EtCetera cannot explain but which nevertheless is not terminal. If this error occurs, please follow the procedure discussed earlier regarding "unknown and/or unrecoverable error".

Unknown command

Error number -1.

This error means that you have a typo.   Check the line and make sure the command is properly spelled.

You cannot use the command specified

Error number 103.

This error occurs when you attempt to use a command which is not supported by your version of EtCetera or your version of Windows.   For example, if you are using Windows 3.00 and do not have the Multimedia Extensions, then

```
PlaySound "CHIMES.WAV"
```

will fail because Windows 3.00 does not support playing sound files without the Multimedia Extensions.

This can also occur if you attempt to use a command which requires that ETCETERA.386 be installed, but you are either running in Real or Standard modes, or you do not have an entry in your SYSTEM.INI file to load this file. These commands are available only with EtCetera 2.10.

This can occur if you use a word for a command which is not supported by EtCetera.   Usually, this will occur because planned (but as-of-yet unimplemented) commands are understood by EtCetera, but EtCetera has not been taught what it is supposed to do when it encounters these commands.

The only way you would normally be aware of such commands is if you are doing no-no's, such as using Norton's DiskEdit program to view the EtCetera file as raw data.

Sample Code

The following list of topics will show you various sample code fragments.

Closing applications with data waiting to be saved
Hiding and unhiding all application windows
How to use DDE commands
How to use DisplayBitmap
Loops, or "Repeating Code Fragments Easily"
Running the same program with different WIN.INI settings
Simulating drag-and-drop

Simulating drag-and-drop

Although EtCetera does permit simulating mouse actions, it is highly recommended that you do not attempt this lightly, as nothing prevents the user from grabbing ahold of the mouse and moving it while you are attempting to move it. Then, when you simulate a click or a double-click, you end up clicking or double-clicking something completely unexpected.

If, however, no users will be around, or you have managed apply a strong electric charge to the mouse to prevent the aforementioned behavoiral problem, then you can easily manage this.

```
Mouse MOVE 100, 100
Mouse LEFT DOWN
Mouse MOVE 300, 300
Mouse LEFT UP
```

This one is best done for your own benefit and not for the user. If, instead, you are trying to demonstrate the use of the mouse and would like to see the mouse in action, you can instead do the following.

```
Mouse MOVE 100, 100
Wait 1 second
Mouse LEFT DOWN
Beep
For #A = 101 to 300
Mouse MOVE #A, #A
Next
Mouse LEFT UP
```

The preceding fragment will instantly move the mouse pointer to (100, 100) and depress the left mouse button. Then EtCetera will gradually move it, one pixel at a time, to (300, 300), where it releases the left mouse button.

Running the same program with different WIN.INI settings

Some programs use WIN.INI settings for configuration information. It is substantially easier to configure an EtCetera batch file for each different configuration and double-click an EtCetera icon than it is to go into the program and reconfigure it each time. The following illustrates how to do this with Microsoft Excel.

```
CD "D:\EXCEL"
SetEntry "Microsoft Excel", "Maximized", "TRUE"
Run "EXCEL"
```

Obviously, the Maximized entry is not particularly important, but it does illustrate the point.

Loops, or "Repeating Code Fragments Easily"

Many of the other samples show this technique. This is very common in many programming languages.

The technique uses the FOR command with NEXT to repeat code many times. For instance, the following will display, successively, the text of title bar of each open window in a dialog box and wait for the user to click OK with each one:

```
TitleFill 100
For #C = 100 to 99 + Lines
Message $[#C]
Next
```

You can place more than a single line between FOR and NEXT. You could instead ask the user if he wishes to continue while displaying each line and, if not, use the BREAK command to end the loop:

```
TitleFill 100
For #C = 100 To 99 + Lines
Ask $R, $[#C] & Ansi(13) & "Do you wish to continue?"
If $R = "No" Then BREAK
Next
```

BREAK always jumps to the line following the next Next command, regardless of how many lines are between the BREAK command and the Next command.

How to use DisplayBitmap

DisplayBitmap can be used to place a bitmap on any window which you can activate. Unless you specifically create the window yourself, however, the application which owns the window will probably soon redraw whatever image it wants in the window, so beware.

Assuming that you want to create your own window to display a bitmap, the following code is an example of how to do this.

```
CreateWindow @A, 100, 100, 300, 300
DisplayBitmap @A, "SAMPLE.BMP", 0, 0
```

Note that you use the same handle (@A) that you used with CreateWindow.

If you want to do it with, instead, Notepad, you could use

```
GetHandle @D "Notepad"
DisplayBitmap @D, "SAMPLE.BMP", 0, 0
```

Using "0, 0" tells EtCetera to place the bitmap's upper left corner at the client coordinates (0, 0), which is the upper left corner of the window's client area. Client area means the area where a program can draw in a window normally. (The program is the "client", which has "leased" some space from Windows to display data for the user, and this space is the client's area, or "client area".) This excludes the title bar, scroll bars, borders, menu bars, and the like.

Unlike many other commands, DisplayBitmap (along with WriteText) use coordinates which are client coordinates (or window coordinates), rather than being screen coordinates, which place (0, 0) at the extreme upper-left corner of your video display.

If you are inclined to try it, you can simulate animation with this command, such as

```
CreateWindow @D, 100, 100, 300, 300
For #A = 0 to 9
DisplayBitmap @D, "SAMPLE" & $#A & ".BMP", 0, 0
Next #A
```

This takes SAMPLE0.BMP, SAMPLE1.BMP, SAMPLE2.BMP, ..., SAMPLE8.BMP, and SAMPLE9.BMP rapidly displays them. If they are successive frames of some image, this will form a crude animation. Of course, with varying CPU speeds and video display cards, you can get varying results. Know your target PC before relying on this.

How to use DDE commands

The DDE commands are specific to the other application with which you wish to communicate. However, just for argument's sake, illustrated here are samples from some better known applications. Note that inclusion here does not indicate a preference or endorsement for the products listed.

```
DDEPeek "Excel", "FINANCES.XLS", "R2C4", $[103]
```

This command will retrieve the text in cell D2 (Row 2, Column 4) and place it in $[103].

```
DDEPoke "Excel", "FINANCES.XLS", "R2C5", $A
```

This command reverses the process, placing the text in $A into cell E2 on FINANCES.XLS. You can even send a formula to Excel, such as the following.

```
DDEPoke "Excel", "FINANCES.XLS", "R2C5", "=SUM(A1:A10)"
```

An example of how to use DDEExec with Program Manager is

```
DDEExec "PROGMAN", "PROGMAN", "[CreateGroup(""Test Group"")]"
```

Note the use of the double quotation marks surrounding the words Test and Group. The name of the group in this case must be enclosed in quotation marks, so a pair of quotation marks is used to tell EtCetera to send quotation marks to Program Manager.

Hiding and unhiding all application windows

This code fragment will hide all open windows:

```
TitleFill 200
For #A = 200 to 199 + Lines
Hide $[#A]
Next #A
```

To unhide all open windows, substitute Unhide for Hide, above.   Unfortunately, this will unhide all open windows, including those otherwise left hidden while an application runs, such as windows which are used by an application to field DDE messages.   Try this technique with an application which supports DDE requests (a DDE server) and observe the results.

Closing applications with data waiting to be saved

The easiest way to close an application with data waiting to be saved is to know beforehand which files it has open and then close them. Since, however, that is not always possible, the following code fragments could prove useful. In all cases, Notepad is the application used.

```
Close "Notepad"
Wait 3 seconds
If IsOpen "Notepad" Then SendKeys "{tab}{enter}"
```