MiContact Centre Office - SDK Technical Manual

NOVEMBER 2015

DOCUMENT RELEASE 6.2

TECHNICAL MANUAL

Mitel

Table of Contents

NOTICE

The information contained in this document is believed to be accurate in all respects but is not warranted by Mitel Networks™ Corporation (MITEL®). The information is subject to change without notice and should not be construed in any way as a commitment by Mitel or any of its affiliates or subsidiaries. Mitel and its affiliates and subsidiaries assume no responsibility for any errors or omissions in this document. Revisions of this document or new editions of it may be issued to incorporate such changes.

TRADEMARKS

MiContact Centre Office - SDK
Release 6.2 - November, 2015

1 What's New

What's New in this Release

Release 6.2

Updated Branding & User Interface

This release of MiContact Center Office sees the product branding change from Customer Service Manager. MiCC Office is the new abbreviation for CSM. Also included in the product is an updated user interface that implements the following concepts:

- Updates to support the latest Windows common controls (buttons, tabs etc)
- Updates to the default color palette used by the client applications
- Updated icons and graphics to bring the product in line with Mitel Branding

Windows Support

MiCC Office Server:

- Windows 8.1 Standard/Professional/Enterprise (64-bit)
- Windows 8 Standard/Professional/Enterprise (64-bit)
- Windows 7 Professional/Ultimate SP1 (64-bit)
- Windows Server 2008 R2 SP1 (64-bit)
- Windows Server 2012 R2 Standard/Essentials/Datacenter Editions (64-bit)

MiCC Office client applications are supported on these versions of Windows:

- Windows 8.1 Standard/Professional/Enterprise (32-bit and 64-bit)
- Windows 8 Professional (32-bit and 64-bit)
- Windows 8 Standard/Enterprise (32-bit)
- Windows 7 Professional/Ultimate SP1 (32-bit and 64-bit)
- Windows Server 2008 R2 SP1 (64-bit)
- Windows Server 2012 R2 Standard/Essentials/Datacenter Editions (64-bit)

vSphere Support

MiCC Office 6.2 includes support for VMare vSphere 6.0 for deploying Virtual MiCC Office appliances and installing MiCC Office on virtual machines. For information about deploying Virtual MiCC Office, see the Virtual Appliance Deployment guide on the Mitel eDocs Web site www.edocs.mitel.com.

SMTP Support

SMTP emailing support has been added to Intelligent Router rules and the Auto Reporter features of Reporter. This allows emails to be sent without having to have a MAPI enabled email client running. MAPI support has been left in the applications for backward compatibility.

SMTP SSL\TLS Support

SSL\TLS support has been added to all areas where SMTP is used; Media Blending, Intelligent Router rules & Auto Reporter. If your mail server supports the SSL\TLS feature then it can be enabled for use with MiCC Office.

Media Blending Add-In Update

The MiCC Office Callviewer plugin for Microsoft Outlook has been given a facelift and has been updated to support the latest versions of Outlook.

Backup Utility Update

A new service based backup utility has been introduced to the MiCC Office Server to automate the process of keeping multiple backups of the solution to help minimize the risk of data loss from hardware or software failure. This new backup utility can store backups on the local server or a network drive.

2 Introduction

Introduction

CallViewer is a module in MiContact Center Office that provides first party call control of a call center agent's phone, as well as displaying information about who is calling or being called. It also allows a user to screen pop a customer's record in the company database based on the caller's details, such as the telephone number. CallViewer comes with several "Ready To Go" solutions pre-installed that will screen pop several common databases such as Microsoft Access, GoldMine, etc.

However, some companies may use a proprietary database for a specific end-solution, or a database that is not supported by CallViewer . In that instance, the end-user would need to use a custom action to integrate with their database. The custom action would take information that CallViewer knows about the call, and then interact with the database using COM, DDE or keystrokes to display the customer's details.

To achieve this CallViewer has a built-in macro language that enables the creation of advanced actions that can perform repeated tasks quickly, based on information entered by the user, or information known by CallViewer . CallViewer Callview also has an Active X control that allows a developer to integrate their application with CallViewer . The Software Developer's Kit (SDK) documents the macro language and the Active X control. CallViewer uses the compiler dongle when a user-defined action needs to be compiled. CallViewer will not use an action until it has been compiled.

3 What Is In the SDK?

What Is In the SDK?

The Software Developer's Kit consists of the following items:

- **This manual**: The manual documents how to create user-defined actions, along with use of the CallViewer macro language and Active X control.
- **The SDK help file**: The help file is installed by the SDK installation program, enabling you to get help on the macro language from within the macro editor.
- **Compiler dongle**: The compiler dongle is a USB key that you must plug into an available USB port when you want to compile an action that you have written. CallViewer will check for the dongle when you save a user-defined action. If the dongle cannot be found, the action is not compiled. An action that has not been compiled cannot be run. The dongle is not needed to run the user-defined actions however.
- **Sample File**s: When you install the SDK, several sample actions are provided as text files to help you get started. There are also some sample Visual Basic applications to help you learn how to use the Active X control.

4 Installing the Software Developer's Kit

Installing the Software Developer’s Kit

The installation of the SDK is very simple, since it is just an additional component to CallViewer . Before installing the Developer SDK, ensure that you have installed CallViewer first. If you forget to install CallViewer , the Developer SDK install will display an error message indicating that CallViewer needs to be installed first.

To install the SDK, insert the Developer SDK CD-ROM into your CD or DVD drive. The installation program automatically starts. After you accept the license agreement, the installation program installs the necessary files to your hard drive.

By default, CallViewer is installed to C:\Program Files\ Contact Center Suite , with CallViewer being installed to a CallViewer subfolder. Most of the SDK will be installed to a folder called SDK from the Contact Center installation folder, although some files will also be installed to the CallViewer folder. Shortcuts are put in the usual Contact Center program group.

Note: If you uninstall CallViewer , the Developer SDK will not be automatically uninstalled. You should use the Developer SDK uninstall routine to uninstall the SDK. Uninstalling the SDK will not uninstall any other element of CallViewer .

5 Creating User-Defined Actions

Creating User-Defined Actions

You create and edit your actions in the Action Manager within . You can access the Action Manager from by right-clicking the tray bar icon and choosing **Actions**.

The Action Manager shows a list of user-defined actions that you have created, along with instances of Ready To Go actions that have been created. You can add a new user-defined action by clicking the **Add** button, and then selecting **User Macro**.

Alternatively, select the action that you want to change, and click **Edit**.

Adding or editing a user-defined action will display the Macro Editor, where you enter the code for your action.

5.1 Macro Editor

Macro Editor

The macro editor is where you enter the code for your user-defined actions. The macro editor consists of a toolbar, and the area where you write your code.

Before you start writing your action, you should decide whether you are going to use the macro language, or an ActiveX scripting language such as VBScript or JavaScript. Although you can choose the language at any time, you will find it easier if you set the language correctly from the start.

You can choose the language for your action from the drop-down box on the toolbar. You can also enter any valid ActiveX scripting language name in the drop-down box, if additional scripting languages are installed on the client computers.

Having chosen the language you want to use, you can start writing your code. See for further information.

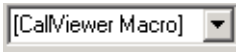
The Editor

The main portion of the Macro Editor is taken up by the area where you enter your code. This works in the same fashion as a standard Windows text editor such as Notepad. The status bar at the bottom of the window will indicate the current line and character position that you are at, which is useful if an error occurs when you run your macro, and you need to locate the problematic piece of code.

The Toolbar

The toolbar at the top of the Macro Editor consists of the following options:

	**Main Menu**: Displays the Main Menu. Information on the Main Menu options is available i the “Main Menu” section below.
	**Save**: Saves the action that you are writing. If you have not saved the action before then you will be prompted for the name to give the action. If the action has been saved already then it will be saved using the existing name. Use the Save As option on the menu if you want to rename the action. For further information on saving actions, see Saving Macros.
	**Import**: This option will prompt you for a text file to import at the current cursor location
	**Insert**: Displays a menu of known functions and variables that can be used in the currentl selected macro language that you are using. The menu contains two sections, **Macros** and **Variables**. The **Macros** part of the menu consists of several sub-menus for different categories of command. Each sub-menu consists of similar macro commands for the category that you have selected. Selecting an item from a menu will insert that macro command at the curre cursor position, and include the default parameters for that macro command. The **Variables** part of the menu consists of several sub-menus for different categories of variable. Each sub-menu consists of similar variables for the category that you have selected. Selecting an item from a menu will insert that variable at the current cursor position. You can also click **Macros** or **Variables** to display the list of macros or variables in a dial Selecting an item in the dialog will display a brief description of that item. Clicking Insert w insert the selected command at the current cursor position.
	**Cut**: Places the highlighted text in the clipboard, before removing it from the editor.

	**Copy**: Places the highlighted text in the clipboard.
	**Paste**: Inserts any text stored in the clipboard at the current cursor position.
	**Current Language:** Specifies which macro language you are writing the action in. You would normally select either Macro or VBScript in this section, although you can enter the name of any ActiveX scripting language that is installed on the company's computers. The language that you select here will affect how compiles the action when save it. If you sele the wrong language, you are likely to get compile errors when you save the action.

The Main Menu

Click the **Menu** button on the toolbar to display a menu containing the following options:

Save	Compiles and saves your current macro using its current name. If the macro has not been saved before, you will be prompted for a name for the macro, as if you had clicked Save As. See Saving Macros for further information.
Save As	Prompts you for a name to save the macro as. This does not create another instance of the macro if it has already been saved; it effectively renames this macro with a new name. See Saving Macros for further information on saving macros.
Import	Displays the standard File Open dialog box to allow you to select a text file that will be inserted at the current cursor position.
Export	Displays the standard File Save dialog box to allow you to save the selected text in the Macro Editor as a text file.
Cut	Places the highlighted text in the clipboard, before removing it from the editor.
Copy	Places the highlighted text in the clipboard.
Paste	Inserts any text stored in the clipboard at the current cursor position.
Close	Closes the Macro Editor. If you have not saved the macro after editing it, you will be prompted to save the macro first. This option is the same as clicking the Windows close button on the Macro Editor window.

5.1.1 Saving Macros

Saving Macros

Before a macro can be used, it must be saved. Saving a macro also compiles it, which displays any errors in the macro syntax. You can save a macro in several ways:

- Click the **Save** button on the toolbar.
- Choose the **Save** or **Save As** options on the main menu.
- Close the Macro Editor after making changes to the macro and not saving. You will be informed that you have not saved, and can choose to continue closing without saving the macro, or save the macro now.

The first time that you save a macro you will be prompted to enter a name for the macro. The name of each macro must be unique; if you enter a name that is already in use, you will not be allowed to continue.

The macro is then compiled and saved. When the macro is compiled, CallViewer will check that the SDK dongle is attached to the computer. The dongle licenses CallViewer to be able to compile macros. If the dongle cannot be located, CallViewer will display an error, and give you the opportunity to insert the dongle before trying to compile the macro again. If there is a syntax error in your code, the macro will not be compiled, but will be saved. You will be informed of the error, and the related line of code will be highlighted.

Note: Although you can close the Macro Editor after saving a macro that has failed to compile, you will not be able to run the macro until it has compiled correctly.

Also: A CallViewer license is required for any CallViewer that tries to execute a user-defined macro.

5.2 Simulation Mode

Simulation Mode

The Simulation Mode allows you to simulate calls and e-mail processing without connecting to the MiCC Office Server . It is a very useful tool for testing your macros with without having to create several calls or e-mails, and without the need to connect to a live system.

To enable Simulation Mode, right-click the CallViewer tray bar icon, and select Options from the menu. On the Call Control tab, select Enable Simulation Mode. You do not need a valid connection to a Contact Center Server when using simulation mode. However, without a valid connection to a MiCC Office Server you will need to plug the SDK dongle into the computer to license CallViewer , because it normally decides on the license level based on a response from the MiCC Office Server . You should plug the dongle into the computer before you enable simulation mode, otherwise you may not obtain a license, resulting in CallViewer not functioning.

The Simulation Window

The Simulation Window can be used to simulate calls and e-mails. The top part of the window contains a toolbar to select common options, while the bottom part of the window, the Call / E-mail Detail area, allows you to configure information about the call or e-mail being simulated.

The Toolbar

The toolbar consists of the following options:

	**Media Type To Simulate**: Selects whether to simulate calls or e-mails. After you select the type media to simulate, the lower half of the window will change to show settings that you can use to simulate that type of media.
	**Update**: Updates CallViewer based on the settings you have specified in the lower half of the Simulation window. If you have specified settings for a new call, then clicking this button would make a new call appear in CallViewer .
	**Delete**: Deletes the call or e-mail based on the current settings in the simulation window.
	**Toggle Answer**: Toggles the Answered state of the given simulated call. If the call is alerting it w be shown as answered, and if it is answered it will return to alerting. If no call is being simulated, click this button to use the settings in the lower half of the simulation window to create a new simulated call.
	**Toggle Hold**: Toggles the Hold state of the given simulated call. If the call is on hold it will be tak off hold, and if the call is not held it will become held. If no call is being simulated, click this buttor to use the settings in the lower half of the simulation window to create a new simulated call.
	**Busy**: Simulates the handset becoming “busy,” i.e., being taken off hook.
	**Idle**: This button mimics the handset going “idle”, i.e., being placed on hook.

5.2.1 Simulating Calls

Simulating Calls

To simulate a call, first ensure that you have selected Calls from the Media Type option on the toolbar. Then fill in the Call / E-mail Detail section of the Simulation window and click Update. The controls in the Flags section can be used to modify the state (held, answered, etc.) of the call.

The “Line / Extension No” field is used to define the call you are changing. If you have created a simulated call on line “100”, and want to simulate a second call, you should change the “Line / Extension No” field to a different value, e.g., “101”, which would result in two calls being shown in – one from device 100, and one from device 101.

Note: If you select a call in the Active Call List window, its current details will be displayed in the Simulation window’s Call / E-mail Detail section.

To remove a simulated call from the display, enter the line/extension number for the call into the Line/Extension No text box, and then click **Delete**.

The call properties shown below can be entered for a simulated call.

/ Digits	The digits or received for the simulated call, which corresponds to the subsequent value of the [**Digits**] macro variable for the current call. For a manual (button) macro, the current call is the selected call in the call list. For an automatic macro the current call is the activating call in the call list that caused the macro to run.
Line / Extension No	The line or extension number for the simulated call, which corresponds to the subsequent value of the [**Line**] and [**Col1**] macro variables for the current call.
DNIS (Col 2)	The DNIS string for the simulated call, which corresponds to the subsequent value of the [**DNIS**] and [**Col2**] macro variables for the current call. Ordinarily, this call property would take one of the following string values for the current call; either the DNIS string found against the number by the distant end, the trunk line description for external non- calls, or the value “[Internal]” if the current call is an internal call.
Col 3 (Import Field 2)	The value held within column 3 of the call list for the simulated call, which corresponds to the subsequent value of the [**Col3**] macro variable for the current call. Normally, column 3 would represent field 2 from the matched record in the telephone data import file that the MiCC Office Server ( ) uses to identify and digits against, although this would only be the case for external calls.
Col 4 (Import Field 3)	The value held within column 4 of the call list for the simulated call, which corresponds to the subsequent value of the [**Col4**] macro variable for the current call. Normally, column 4 would represent field 3 from the matched record in the telephone data import file that the MiCC Office Server ( ) uses to identify and digits against, although this would only be the case for external calls.
Col 5 (Import Field 4)	The value held within column 5 of the call list for the simulated call, which corresponds to the subsequent value of the [**Col5**] macro variable for the current call. Normally, column 5 would represent field 4 from the matched record in the telephone data import file that the MiCC Office Server ( ) uses to identify and digits against, although this would only be the case for external calls.
Col 6 (Import Field 5)	The value held within column 6 of the call list for the simulated call, which corresponds to the subsequent value of the [Col6] macro variable for the current call. Normally, column 6 would represent field 5 from the matched record in the

	telephone data import file that the MiCC Office Server ( ) uses to identify and digits against, although this would only be the case for external calls.
Col 7 (Import Field 6)	The value held within column 7 of the call list for the simulated call, which corresponds to the subsequent value of the [Col7] macro variable for the current call. Normally, column 7 would represent field 6 from the matched record in the telephone data import file that the MiCC Office Server ( ) uses to identify and digits against, although this would only be the case for external calls.
Digits	A value representing the actual digits or number (depending on the telephone system) for the simulated call, which corresponds to the subsequent value of the [**DDIDigits**] macro variable for the current call.
Call Serial No	The unique serial number for the simulated call, which corresponds to the subsequent value of the [**CalLSerialNo**] macro variable for the current call.
Answered?	Use the **Yes** button to indicate that the given call is answered, or the **No** button to indicate that the call is alerting.
Direction	Use the **In** button to indicate that the given call is inbound, i.e., another caller calling this device, or use the **Out** button to indicate that the given call is outbound, i.e., a call from this device.
Internal?	Use the **Internal** button to indicate that the given call is internal, i.e., to another device on the telephone system, or use the **External** button to indicate that the given call is external, i.e., to an outside number.
Held?	Use the **Yes** button to indicate that the given call is currently on hold, or use the **No** button to indicate that the given call is currently active (not on hold).
Tel.No.Match?	Use the **Yes** button to indicate that the given telephone number has been identified against the Telephone Number Import file, or use the **No** button to indicate that it has not. A telephone number that has been matched against the Import file is one that can be screen popped using information from Import Fields 2 to 6, since it means that the Import Fields contain valid information. If the telephone number has not been matched, it means that the Import Fields do not necessarily contain valid information.
Received?	Use the **Yes** button to indicate that was received for this call, or use the **No** button to indicate that was not received for this call.

Note: It is possible to configure the flags so that they do not make sense, e.g., no valid telephone number in / Digits, but the Received flag set to **Yes**. This is something to be aware of when testing your user action using the Simulation Window.

5.2.2 Simulating E-mails

Simulating E-mails

To imitate an e-mail being processed that was routed to the associated extension by media-blending Rule, fill in the E-mail Details section (the area below the buttons) of the Simulation window and click **Update**.

To simulate the end of e-mail processing, click **Delete**. Unlike for calls, you can simulate only one e-mail message being processed at a time.

The following properties can be entered for a simulated e-mail:

E-mail From (Address)	The actual e-mail address of the sender, e.g., “customer@company.com”, which correspond to the subsequent value of the [**EmailFromAddr**] macro variable.
E-mail From (Display Name)	The description assigned against the e-mail address of the sender, which corresponds to the subsequent value of the [**EmailFromName**] macro variable. The description can be defined the original sender of the e-mail, or the local e-mail server depending on the type of server and/or the server’s particular configuration.
E-mail To (Address)	The e-mail address of the inbound mail queue that the e-mail was sent to and distributed via using media-blending rule. This message property corresponds to the subsequent value of th [**EmailToAddr**] macro variable.
E-mail To (Display Name)	The description assigned against the e-mail address of the inbound mail queue that the e-ma was sent to and distributed via using media-blending rule. The description can be defined by the original sender of the e-mail, or the local e-mail server depending on the type of server and/or the server’s particular configuration. This message property corresponds to the subsequent value of the [**EmailToName**] macro variable.
Subject Text	The subject line of the e-mail message (255 characters maximum length) as seen by the use in their local e-mail client against the corresponding e-mail. This value corresponds to the subsequent value of the [**EmailSubjectText**] macro variable.
E-mail Tag	The internal reference code assigned by to the e-mail message, which is unique in real-time all active e-mail messages being queued by any media blending Rule. The property corresponds to the subsequent value of the [**EmailTag**] macro variable.
Queue (Hunt Group)	The device number of the hunt group, which corresponds to the media blending queue that t e-mail message arrived via, as defined within ’s hunt group configuration. The property corresponds to the subsequent value of the [**EmailGrpQ**] macro variable.
Original Tag	The internal reference number of the e-mail message assigned by the instance that downloaded the message. The property corresponds to the subsequent value of the [**EmailTagOrig**] macro variable. It is this value that can be used to identify the actual e-mail message in the user’s e-mail clie (e.g., Microsoft Outlook). The e-mail message will exist in the user’s inbox and will contain th tag at the beginning of the message’s subject line. The display format of the tag’s value is th hex representation of the number and is always padded with leading zeros so that the tag’s length is always 8 characters long, as well as the tag value being encapsulated using the character sequences “#[” and “]#”. For example, an e-mail message with an original tag valu of 10 would be displayed in the subject line as “#[0000000a]#”. As an illustration, here is an example subject line for an e-mail message that has been distributed using media-blending Rule: “#[00000002]# Attn Technical Support”.
Size (Bytes)	The size in bytes of the e-mail message, which corresponds to the subsequent value of the

	[**EmailSize**] macro variable.

Note: If you have enabled the Simulation window but you have accidentally closed it, you can open the Simulation window again by selecting the “Show Simulation Window” menu item on the main menu. Right-click the tray bar icon to access the main menu.

Also: While the Simulation window is open, you can select a call within the call list which will appropriately update the Simulation window’s Call / E-mail Details section.

6 CallViewer Macro Language

CallViewer Macro Language

CallViewer provides a programming interface where the user can write native CallViewer macro scripts that can interrogate and drive CallViewer , e.g., making a call, activating a window, etc. This section will give a brief introduction to the macro language, explaining what a macro script consists of, and then later on introducing some of the more commonly used commands. A complete reference for the macro language is available in the Reference topics.

Macro Scripts

Expressions in Macros

ANSI References in Macros

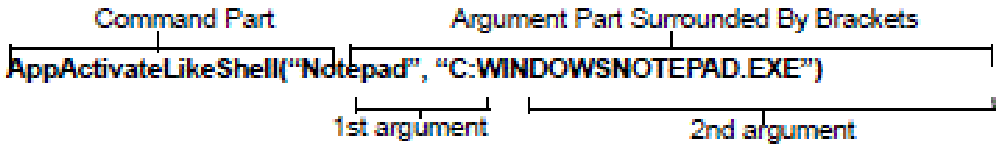
6.1 Macro Scripts

Macro Scripts

A macro script consists of a list of macro commands. A macro command is a lot like a verb in a modern language – it describes something that needs to be done. Macro commands can also accept parameters that define how things should be done. Parameters are a lot like nouns in a modern language.

When writing a macro, each separate macro command statement should be on a separate line. You may insert empty lines if you want to make the macro script more readable. You can also add comments to your script to help you remember what it does by prefixing a line with a single quote.

Each native macro command has a Command part (the verb) and, if required, one or more Arguments (nouns).



The Command Part identifies the macro's own individual function. The arguments allow you to specify information that the macro command needs to perform its particular operation. If the macro command requires more than one argument, a comma must separate each of these.

Specifying several macro commands together forms a macro script, and each macro command will be performed in turn, waiting for the last one to complete before proceeding to the next.

You can write a macro directly as VBScript, JScript, or JavaScript, in which case the rules for writing such a macro are different. However, programming in such languages is intended only for professional programmers, and so no entry-level introduction will be considered here.

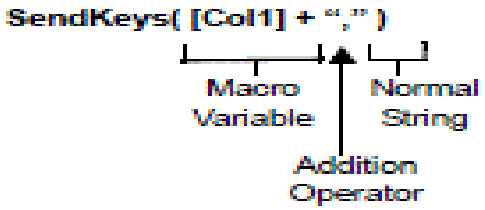
Expressions in Macros

ANSI References in Macros

6.2 Expressions in Macros

Expressions in Macros

Some macro commands accept arguments that can be built up from expressions. An expression is a special combination of keywords and operators that returns a result at runtime, as illustrated in the following example.



The above example shows the **SendKeys** macro command statement. This macro command can be used to send a keystroke sequence to the currently active application window.

The **SendKeys** macro has one argument, which specifies the keystroke sequence that will be sent. In this example, the argument is an expression ([Col1] + ", "), which at runtime would yield the value from column 1 of the selected call in the call list, with a comma and space character appended to the end. [Col1] is a Macro Variable. There are many of these, and each allows you to access values that can change during macro runtime. See Macro Variables for more details.

An expression is a combination of constants, macro variables, and mathematical operators (i.e., +, -, *, /, etc.). For example, the expression (2 + 5) * 3 would return 21.

The + operator also operates as a way of adding strings together, e.g., "The " + "cat sat" + " on the mat." would return "The cat sat on the mat.".

Macro Scripts

ANSI References in Macros

6.3 ANSI References in Expressions

ANSI References in Expressions

When you are specifying information to a native macro command by writing a string value into one of it's arguments, there are some characters that cannot easily be specified. For instance, the characters "a" to "z", "0" to "9" etc. are easily specified because you can insert them easily by pressing the corresponding key on the keyboard.

In contrast, control characters such as Escape, Carriage Return, Line Feed, etc. are not so easily inserted because pressing the **Escape** or **Enter** keys on the keyboard, performs an action in the user interface instead of actually inserting a character.

However, there is a way to specify any ANSI-compatible character in your macro string arguments.

Control characters can have a special meaning to applications. For example, the clipboard interprets a Carriage Return (ANSI character 13) followed by a Line Feed character (ANSI character 10) as the start of a new line within text.

Within macro command arguments, string (character) expressions may be constructed using ANSI character references inside "curly" bracket parentheses. Placing a number within the parentheses specifies the actual ANSI text character:

```
"This Line" + {13} + "Next Line"
```

Putting a space before the closing bracket followed by another number specifies that the ANSI equivalent of the specified number will get repeated that many times (e.g., {10 12} = 12 line feed characters in a row).

The following is an example of a macro command with one argument built from an ANSI character expression:

```
' Copy text with carriage return and line feed
' characters appended to the end of string.
ClipboardSetText(" " + {13} + {10})
```

Macro Scripts

Expressions in Macros

6.4 Line Labels and Conditional Jumps

Line Labels and Conditional Jumps

CallViewer 's macro language allows conditional execution based on the state of another application, or the evaluation of expressions. This is supported through the use of line labels and conditional jumps.

For instance, you might want to automate a specific task only when a call is from a particular location or for calls on an individual DID number.

The macro language supports conditional branching to specific points in the macro script defined by line labels through the use of "Goto" macro command statements. The line labels are text strings with no space or punctuation characters ending in a colon. Each line label must be unique in the macro in which it is used. Line labels are not case sensitive.

The first argument in any of the "Goto" macro command must refer to a valid line label with the current macro script.

For example, the following macro would only show a message box if a call was received with DID digits of "5000":

```
' Branch if DID digits equal "5000"
GotoIfStrValue("labelDDI5000", [DDIDigits], "5000", 0)
Goto("labelExit") ' Exit if DID digits are not "5000".
labelDDI5000:
' Show message window.
MessageBox(" DID 5000 call", 0, " CallViewer ")
labelExit:
```

6.5 Macro Variables

Macro Variables

Native macro command argument expressions can refer to macro variables that are replaced at runtime with the related value. There are macro variables for call properties, such as the of the current call, as well as variables for other information, such as the current time, or the title of the currently active application.

Macro variable names are enclosed in square brackets, e.g., [**Digits**]. This differentiates them from constants, such as text or numbers. For instance, the following would be a valid expression:

```
"Tel " + [Digits]
```

When the macro is run this would evaluate as...

```
Tel
```

...if the currently active call was from " ".

The following is an example of a macro command with 1 argument built from an expression that uses a variable reference:

```
' Copy line number of current call with carriage return
' and line feed characters appended to the end of string.
ClipboardSetText("Line Number: " + [Line] + {13} + {10})
```

A complete list of macro variables is available in the Reference topics.

6.6 Screen Popping With Actions

Screen Popping With Actions

Screen Popping is where will automatically locate a customer's details in your company database using information on the currently active call. If you are using a proprietary database, then you will probably need to create a user action to screen pop your database. This section explains the common elements of a screen popping action, and introduces some macro commands that will achieve a screen pop.

Any screen popping action will generally follow the following format:

- Check the application
- Check the call
- Search the application for the data

Each of these elements is described in the topics linked above. However, a successful screen pop will also depend on the data integrity of the application to be screen popped, and the information contained in the Import file.

6.6.1 Checking the Application

Checking the Application

Before attempting to drive an application, the action should ensure that the application is at an appropriate point to be driven. If dialog boxes are open, or the application is in an unexpected view, the screen pop may not work.

In some situations you can work around the problem, e.g., switching to a particular view, or closing an open dialog box. However, it is sometimes necessary to limit the macro so that it has to expect the application to be in a given state before screen popping. In such a situation, the customer or user must be informed of the limitation.

At a minimum, you must ensure that the application is open. This is achieved with the **GotoIfAppActive** or **GotoIfAppActiveRight** commands, which jump to a specific point in the action based on whether a window is open or not. The following code sample checks if Microsoft Access is open, ending the action if it isn't.

```
' Jump to AccessOpen if window "Microsoft Access" is open
GotoIfAppActive("AccessOpen","Microsoft Access",1)
' Access is not open, so quit the macro
End
AccessOpen:
' Access is open so continue
```

Note: If you want to check for an application using the end part of the window title, use the **GotoIfAppActiveRight** macro.

6.6.2 Checking the Call

Checking the Call

After checking that the application to be screen popped is open, you should check the state of the call to ensure that it is appropriate for screen popping.

The first thing to do is check that calls are present at the current extension. This is achieved with the **GotoIfNoCalls** command, which jumps to a specific point in the action based on the number of calls present at the extension. The following code sample checks that only one call is present at the extension, ending the macro otherwise.

```
' Jump to OneCall if only one call
GotoIfNoCalls(“OneCall”,1,0)
' Dropped through to this line, so not one call, so end
End
OneCall:
' Only one call at the extension, so continue...
```

After checking the number of calls, check the status of the call, e.g., answered, held, contact identified, etc. This is achieved with the **GotoIfCallType** command. This command accepts a call status flag to compare against the current call, and jumps to a specific point in the action based on whether this flag is set or not. You may have to call this command more than once to completely check the status of call. The following code checks the call status, first to see if the contact is identified, and second to ensure that the call is inbound.

```
' Check that the contact is identified, jumping to
' ContactFound if it is
GotoIfCallType(“ContactFound”,8,0)
' Contact not found, so end
End
ContactFound:
' Check that the call is inbound, jumping to CallInbound
' if it is
GotoIfCallType(“CallInbound”,3,0)
' Call is outbound, so end
End
CallInbound:
' Continue with the screenpop
```

Finally, you should check the data that you will be screen popping from. How this is done depends on how you intend to screenpop the data. Usually, the Import should be configured so that every telephone number to use in screen popping will be associated with a unique identifier with which to screenpop. Let's assume that in the Import, the sixth field will contain an identifier with which we can screenpop the application. Therefore, we need to check that the sixth field contains the appropriate data. The following code checks field 6 to ensure it contains data.

```
' Check that the sixth field (variable [Col7]
' since [Col1] is the line number the call came
' in on, and [Col2] is the DNIS for the line)
' contains data. Jump to GotData if it does
GotoIfStrValue(“GotData”,[Col7],“”,1)
' Col7 is blank, so we cannot screenpop
End
GotData:
' Continue here
```

6.6.3 Searching the Application for the Data

Searching the Application for the Data

How you actually screen pop a record depends completely on the application you are screen popping. In essence, there are two ways of screenpopping - with keystrokes, ODBC, and with Dynamic Data Exchange (DDE). See also Other Alternatives.

6.6.4 Keystrokes

Keystrokes

Screen popping with keystrokes involves activating the application, sending keystrokes to it to display a search dialog, and then more keystrokes to enter the data to be searched for.

First you must activate the application, or a window belonging to the application, so that keystrokes can be passed to the window. This is achieved with the **ActivateApp** macro command.

ActivateApp activates the top-most window that has a caption that matches the text provided. The command is useful because it allows wildcards to be used in the specified text. For example, the command:

```
ActivateApp("Microsoft Word*")
```

would activate the first window that started with "Microsoft Word". Alternatively, the command:

```
ActivateApp("*Notepad")
```

would activate the first Notepad window (since Notepad puts the open filename at the start of the caption, and not at the end like Word does, e.g., "My Notepad File – Notepad").

Having activated the application appropriately, you can then screen pop it. This is achieved by sending keystrokes to the window. Normally these keystrokes would display some form of Search window, into which the data to search for would be entered, again by sending keystrokes to the active window.

Two macro commands are available for sending keystrokes to a window - **SendKeys** and **SendKeysEx**. Both accept a string that contains the keystrokes to be sent. SendKeysEx however also accepts a delay to use between each keystroke.

The keystroke string can be composed of normal ASCII characters, e.g., "These are my keystrokes," or can contain special characters enclosed between { and }. This is normally used for sending non-ASCII characters, such as the End, Escape and cursor keys. For example, the following statement enters some text into Notepad, and then selects the entire text:

```
ActivateApp("*Notepad")
SendKeysEx("^{HOME}This is a test~+{UP}",0)
```

See **SendKeys** for information on the non-ASCII characters that can be used between { and }, as well as the meaning of the modifiers ^, %, and +.

Generally speaking, you should normally use SendKeys to send keystrokes to normal Windows applications, and SendKeysEx to send keystrokes to DOS applications or Win32 console applications running in a window. However, there can be exceptions to the rule that depend entirely on the application being screen popped.

Screenpop Example

The following action screenpops a fictitious DOS application, using the SendKeysEx command.

```
ActivateApp("*Fictitious DOS Application*")
' We assume that the application is at the main menu.
' We must select option 2, to browse the customer records
SendKeysEx("2",0)
' The customer records can be searched by pressing
' Alt and S
SendKeysEx("%s",0)
' Now enter the account reference, which we retrieve from
' the last field of the MiCC Office import ([Col7])
SendKeysEx([Col7]+"{ENTER}",0)
' The application will now display the record, or will
' display an error message to the user, so they can
' create a new record
```

For more information regarding these commands, see the Macro Commands topics in the Reference section.

6.6.5 Dynamic Data Exchange

Dynamic Data Exchange (DDE)

Dynamic Data Exchange (DDE) is a method of sending a foreign application commands and data. For instance, using DDE it is possible to execute menu commands and send data directly to an application. The type of things you can do with DDE depends on the application you are trying to communicate with.

Generally, all Microsoft Office products have excellent DDE facilities. For example, you can open a form in Microsoft Access and navigate to a required record all by using DDE.

Two applications using DDE to exchange data are said to be involved in a DDE conversation.

can invoke more than one conversation (up to six) with the same application at the same time, but each conversation occurs on a different channel. You can initiate DDE conversations in using the macro language.

The application that initiates a conversation is called the destination application, or DDE client, for that conversation. The application that responds to a DDE client is called the source application, or DDE server, for that conversation. Some applications can initiate a conversation with an application on one channel and respond to another application on another channel.

The following topics provide more details on using DDE.

Identifying The Application

Application Names

Topics

Items

Starting a DDE Conversation

Supplying Data To Other Applications

Obtaining Data From Another Applications

Sending Commands To Other Applications

Closing DDE Conversations

Typical DDE Command Sequence

Example DDE Conversation

6.6.6 Identifying The Application

Identifying The Application

Before you can have a conversation with an application, you need to be able to identify which application you are going to communicate with, and what information you will be asking it about.

Application Names

Topics

Items

Starting a DDE Conversation

Supplying Data To Other Applications

Obtaining Data From Another Applications

Sending Commands To Other Applications

Closing DDE Conversations

Typical DDE Command Sequence

Example DDE Conversation

6.6.7 Application Names

Application Names

Every Windows-based application that can participate in DDE conversations has a unique application name. For example, GoldMine's name is "GOLDMINE," and Microsoft Access uses "MSAccess."

To get the application name of another application, see the documentation for that application. The application name is usually the name of the executable file for that application, without the .EXE extension.

Identifying The Application

Topics

Items

Starting a DDE Conversation

Supplying Data To Other Applications

Obtaining Data From Another Applications

Sending Commands To Other Applications

Closing DDE Conversations

Typical DDE Command Sequence

Example DDE Conversation

6.6.8 Topics

Topics

A topic defines the subject of the DDE conversation and represents some unit of data meaningful to the DDE server application. For most applications that operate on files, this is a file name. Some possible topics are a Microsoft Excel worksheet name, for example **`ORDER.XLS`**, or the name of a Microsoft Access database.

Generally, when a topic refers to a file, that file must be open for the DDE server to respond to a conversation initiated about that topic.

Identifying The Application

Application Names

Items

Starting a DDE Conversation

Supplying Data To Other Applications

Obtaining Data From Another Applications

Sending Commands To Other Applications

Closing DDE Conversations

Typical DDE Command Sequence

Example DDE Conversation

6.6.9 Items

Items

An item is a reference to a piece of data (such as a range of cells in a Microsoft Excel worksheet, or a database object in a Microsoft Access database) that can be exchanged between two applications. An item refers to a specific element of a topic.

You need to specify the Application Name and Topic to be able to open a conversation, but having opened a conversation you deal with Items relating to the Topic of the conversation.

Identifying The Application

Application Names

Topics

Starting a DDE Conversation

Supplying Data To Other Applications

Obtaining Data From Another Applications

Sending Commands To Other Applications

Closing DDE Conversations

Typical DDE Command Sequence

Example DDE Conversation

6.6.10 Starting a DDE Conversation

Starting a DDE Conversation

To begin a DDE conversation, you must specify two things:

- The name of the server or source application to talk to.
- The topic of the conversation.

When a server receives a request for a conversation about a topic it , it responds by opening a channel. Once established, a conversation cannot change topics or applications. To begin a conversation with a different server or with the same server about a different topic, you must start a new conversation on a different channel. This does not affect the conversation on the first channel; you can either end the first conversation or continue it.

After a conversation has begun, the two applications exchange messages concerning particular items. An item is usually a reference to a piece of data contained in the topic. Items can vary from topic to topic, and each server can different topics.

To initiate a DDE conversation with another application, use the **DDEOpen** macro command. For example, the following sample will initiate a conversation on channel 1 with Microsoft Excel about a loaded worksheet named “ORDERS.XLS”:

```
DDESetAppName(1, "Excel") ' Set application name.
' Set topic name for DDE conversation.
DDESetTopic(1, "Orders.xls")
DDEOpen(1) ' Start DDE conversation.
```

If **DDEOpen** is not successful in initiating a conversation with the specified application an error occurs. An error can occur if the specified application is not already running, or if the specified application is running but does not the specified topic.

Identifying The Application

Application Names

Topics

Items

Starting a DDE Conversation

Supplying Data To Other Applications

Obtaining Data From Another Applications

Sending Commands To Other Applications

Closing DDE Conversations

Typical DDE Command Sequence

Example DDE Conversation

6.6.11 Supplying Data To Other Applications

Supplying Data To Other Applications

Data can be provided to another application in a DDE conversation using the **DDEPoke** macro command. For example, if you have initiated a conversation with Microsoft Excel using a worksheet as a topic, you can place a new value in the upper-left cell of the worksheet using:

```
DDEPoke(1, "R1C1", "100")
```

Microsoft Excel interprets “100” as the numeric value 100.

DDEPoke can be interpreted as saying “store this value against this item.”

Identifying The Application

Application Names

Topics

Items

Starting a DDE Conversation

Obtaining Data From Another Applications

Sending Commands To Other Applications

Closing DDE Conversations

Typical DDE Command Sequence

Example DDE Conversation

6.6.12 Obtaining Data From Another Applications

Obtaining Data From Another Applications

Data can be requested from another application in a DDE conversation using the **DDERequest** macro command. The collected data returned by the DDERequest macro statement is stored in the appropriate [**DDE*n***] macro variable, where n corresponds to the specified DDE channel used (e.g., [**DDE1**] for channel 1, [**DDE2**] for channel 2... etc.).

For example, if you initiate a conversation with the System topic for any valid DDE application name, you can retrieve a list of all topics currently supported by that application:

```
DDERequest(1, "Topics")
```

If the specified channel does not refer to an active DDE conversation, then an error is generated. You will also get an error if the item was not by the application.

Identifying The Application

Application Names

Topics

Items

Starting a DDE Conversation

Supplying Data To Other Applications

Sending Commands To Other Applications

Closing DDE Conversations

Typical DDE Command Sequence

Example DDE Conversation

6.6.13 Sending Commands To Other Applications

Sending Commands To Other Applications

You can use a DDE conversation to send a command string to another application, although not all applications accept command strings sent in this way. Some, such as Microsoft® Excel® and Word, accept any command string sent to them and carry out that command as if they had been running a Word Basic or Visual Basic macro command.

You send a command string to another application using the **DDESendCmd** or **GotoIfDDESendCmd** statements.

For example, you can send a command to Microsoft Excel that makes it open a worksheet:

```
DDESendCmd(1, "[OPEN(""ORDERS.XLS"")]")
```

The use of square brackets surrounding the "OPEN" command is a Microsoft Excel convention. Microsoft® Access® and Word® for Windows also employ this convention. Other applications may use other conventions, so check your application's documentation for details. The double quotation mark characters are necessary to embed quotation mark characters in the string, which is required by Microsoft Excel.

Identifying The Application

Application Names

Topics

Items

Starting a DDE Conversation

Supplying Data To Other Applications

Obtaining Data From Another Applications

Closing DDE Conversations

Typical DDE Command Sequence

Example DDE Conversation

6.6.14 Closing DDE Conversations

Closing DDE Conversations

When you are finished exchanging data with another application in a DDE conversation, you must close that conversation with **DDEClose**. When execution reaches the end of a macro script all open DDE conversations are closed anyway.

Identifying The Application

Application Names

Topics

Items

Starting a DDE Conversation

Supplying Data To Other Applications

Obtaining Data From Another Applications

Sending Commands To Other Applications

Typical DDE Command Sequence

Example DDE Conversation

6.6.15 Typical DDE Command Sequence

Typical DDE Command Sequence

A typical sequence of events in a DDE conversation is as follows:

1. Set DDE parameters on a channel using **DDESetTimeOut** and **DDESetTimeOutWarningOff**.
2. Set the DDE application name for the conversation using **DDESetAppName**.
3. Set the topic name for the conversation using **DDESetTopic**.
4. Start the conversation using **DDEOpen**.
5. Send commands or data to the application using **DDESendCmd** or **DDEPoke**.
6. Close the conversation using **DDEClose**.

Identifying The Application

Application Names

Topics

Items

Starting a DDE Conversation

Supplying Data To Other Applications

Obtaining Data From Another Applications

Sending Commands To Other Applications

Closing DDE Conversations

Example DDE Conversation

6.6.16 Example DDE Conversation

Example DDE Conversation

The following native macro example finds the company record details in a Microsoft Access database (**CONTACT.MDB**) for the selected call.

First, a DDE conversation is initiated with Microsoft Access using the name of the database as the topic name. Then the appropriate form is opened in the database and the focus is set on the telephone number field. A search is then performed using the telephone number of the caller at the distant end of the trunk line.

```
' Set the focus to Microsoft Access.
AppActivateLike(“Microsoft Access”)
YieldToOS
' Start a DDE conversation with MS Access.
DDESetTimeOut(2, 5)
DDESetTimeOutWarningOff(2, 1)
DDESetAppName(2, “MSAccess”)
DDESetTopic(2, “CONTACT”)
DDEOpen(2)
' Open the Company form in the Access database.
DDESendCmd(2, “[OpenForm ”“Company”“]”)
' Set the focus to the telephone number field.
DDESendCmd(2, “[GotoControl ”“Telephone”“]”)
' Find the record for the caller’s number.
DDESendCmd(2, “[FindRecord ”“” + [Digits] + “”“]”)
' Close the DDE conversation.
DDEClose(2)
```

For this example to work, Microsoft Access needs to be running already. However, since the exact telephone number of the caller is used to attempt to find the record, no guarantee can be made that the correct record will be found because the database may have the telephone number stored in another format (e.g., with spaces or not phone day compatible).

Note: Generally, it is better to use another piece of information than the telephone number to find data in another application. For example, an account number, contract number or sales lead number would be far better. However, for this method to work, the required information needs to be provided in the Import file.

Identifying The Application

Application Names

Topics

Items

Starting a DDE Conversation

Supplying Data To Other Applications

Obtaining Data From Another Applications

Sending Commands To Other Applications

Closing DDE Conversations

Typical DDE Command Sequence

6.6.17 ODBC

ODBC

The Callviewer macro language added support for ODBC in version 4.1. ODBC (Open Database Connectivity) support allows the language to perform database queries against a variety of database types using ODBC drivers supplied with Windows. The ODBC commands in the macro language allow you to directly request information from a given database. However, you cannot use ODBC to make an application display information. You would need to use keystrokes or DDE to display the appropriate data in the application.

Since the ODBC commands are directly querying the customer database, there are several pieces of information you will need for the commands to function:

- The connection details for the ODBC driver.
  This tells ODBC which driver to use, and where the database is stored.

- Username and password.
  Most databases require a username and password to access the information. This is typically provided in the connection details.

- Database query.
  The macro language performs queries on the selected database. You will need to know the query you intend to perform, which invariably requires knowledge of the database structure, such as which tables to include in the query, and the field names used in those tables.

Opening and Closing a Database

To open a database using the macro language, you will use the `ODBCOpen` command. This takes a connection string to define where the database is, and a query string, to define what data you want to query in the database.

It is also a good idea to check that your query has returned some results, since trying to read results when there are none will result in errors!

Before your macro finishes, it should close the database with `ODBCClose`.

The following example opens a sample SQL Server database, and checks that it has some valid results.

```
' Open the database. For clarity, we'll store the connection
' string in the [Data1] variable, and the query in the [Data2]
' variable. The connection string states that the SQL Server is
' on the "DBPC" computer and that the database name is "Pubs", ' using the default SQL Server
' username and password.
DataSetStr(1, "Driver={SQL Server};Server=DBPC;Database=pubs;Uid=sa;Pwd=;")
' The query string is going to ask for all the Customer
' contacts whose telephone number matches the CLI of the active
' call.
DataSetStr(2, "SELECT * FROM Customer WHERE Customer.Phone='" + [Digits] + "';")
' Now open the database…
ODBCOpen(1, [Data1], [Data2])
' Check to see if we got any results. If we didn't, we'll jump
' to the end of the macro
GotoIfNoRecords("NoRecords", 1, 0)

' We got here, so we must have records. Now we'd process our
' database…
```

```
NoRecords:
' Here after we just clean up after ourselves.
ODBCClose(1)
```

More examples of ODBC connection strings can be found later on in the reference information for `ODBCOpen`. You can also contact the database supplier to find out information on the connection string needed. Typically the connection string specifies the driver name to use, and then custom properties to define the location of the database.

Moving Between Records

Having opened the database, it is necessary to navigate the results. When you first open a database you are not always at a particular record, so you must assert the record position.

You use the `ODBCMove` command to set the record position to either the beginning, end, or a given record number. You can also use it to quickly advance to the previous or next record.

As well as being able to set the record position, you also need to be able to decide where in the list of results you currently are. This is achieved with the [`ODBCPosn`] macro variable. If the variable returns -1, you are after the last record; if it returns 0 you are before the first record. In other words, checking for -1 means you cannot proceed any further forward, and checking for 0 means you cannot proceed any further back.

The following sample illustrates all the records in the results until it gets to the end of the records. It is assumed that the ODBC connection has already been opened on ODBC channel 1.

```
' Move to the first record in the results
ODBCMove(1, 0, 0)
' Now process each record til we get to the end
' The following label will be used to define the start of
' processing that we will jump back to if more records
' are available.
ProcessRecords:
' Here we would perform an action on the current record
ODBCGetField(1, "Name", 1)
MessageBox("The name is " + [Data1], 48, "Sample")
' Now we move to the next record
ODBCMove(1, 3, 0)
' Now check if we have reached the last record, and if we
' haven't, jump back to 'ProcessRecords' to process the next.
' Effectively, If [ODBCPos1] <> -1 Then Jump
GotoIfNumValue("ProcessRecords", [ODBCPos1], -1, 1)
' If we reach here, we've processed all the records
```

Reading a Value

Having moved to a record you will probably want to read a value from that record. This is achieved using the `ODBCGetField` command. The command reads a given field name from the current record, into the required [`Datan`] macro variable, where it can subsequently be processed.

In the example in the previous section, the line `ODBCGetField(1,"Name",1)` reads the "Name" field from the current record into the [`Data1`] variable.

Setting Values

You can also use the macro language to update fields in the current record. This is achieved using the `ODBCSetFieldStr` (to update a string-based field) and `ODBCSetFieldNum` (to update a numerical field).

The following sample code shows a "Count" field being read, and then updated by incrementing it.

```
' Get the current value of "Count" into [Data11]
ODCBGetField(1, "Count", 11)
' Now store it back in "Count", but one greater
ODBCSetFieldNum(1, "Count", [Data11] + 1)
```

Error Handling

The Callviewer macro language normally halts execution if an error occurs, and displays information about the error to the user. This is generally a good idea when performing a keystroke macro, since ignoring an error could lead to keystrokes being sent to the wrong application, and then all sorts of strange things could occur.

However, with ODBC, it is often better to allow the macro to check to see if an error was produced and then handle the error itself. This is achieved using the `SetErrorsFatal` command. By default, all commands that generate an error will halt macro execution. However, by using the `SetErrorsFatal` command, this can be stopped.

Once fatal error handling is switched off, then it is up to the macro to check the [`ErrorNum`] and [`ErrorDesc`] macro variables to decide if the last operation caused an error. The macro can then either fail (if the error was truly fatal), or perhaps work around the error with some alternative code. This special error handling only applies to ODBC, DDE, and File handling commands however.

The following sample code shows how error handling might be performed:

```
' Switch off error handling
SetErrorsFatal(0)
' Now try and update a field
ODBCSetFieldStr(1, "LastContactDate", [ShortDate])
' If the [ErrorNum] value isn't 0, then an error occurred
' so jump to the ODBCError label.
GotoIfNumValue("ODBCError", [ErrorNum], 0, 1)
' otherwise it was successful, and we can continue
.
.
.
ODBCError:
' If we got here, an ODBC error occurred
MessageBox("Error: " + [ErrorDesc], 48, "Stop!")
```

6.6.18 Other Alternatives

Other Alternatives

Driving third-party applications using keystrokes is relatively straight forward, but tends to only work if you can guarantee where the user is in the application before the keystrokes are passed.

DDE can circumvent such issues, but obtaining information on DDE items and commands for an application can be quite difficult.

ODBC can be a better alternative, but requires knowledge of the database being queried, and doesn't drive the application's user interface, so can't be used to screen pop.

Another alternative is to use the OLE or COM interface for an application. This is a programmatic interface, similar to the interface provided by the Link Control, which allows a script to drive the given application. Such interfaces are more readily available than DDE information, but they do require that you use VBScript rather than the Macro Language, and they are too involved to be considered in this documentation.

The table below suggests the most appropriate method of integration for several integration scenarios.

Application	Integration Methods
Windows-based and DDE Supported	Use the **SendKeys** macro command where possible. Use DDE for special functionality that cannot be easily accessed using keystrokes.
Windows-based, DDE Not Supported	Use the **SendKeys** macro command.
Windows-based, ODBC Supported	If ODBC is supported but not DDE, then use the ODBC support to process the database, but **SendKeys** or Automation may be required to drive the application's user interface.
MS-DOS Or Console-based	Use the **SendKeysEx** macro command. If this method is unsuccessful, you might try “copying” the keystrokes to the clipboard using the **ClipboardSetText** macro and “pasting” them into the application using the MS-DOS/Console’s System menu.
OLE Automation-supported Only	If the application supports OLE Automation or COM, then you might consider writing a macro script using VBScript and the Link Control (although this is only for the very experienced Software Developer).

6.7 Call Control With Actions

Call Control With Actions

The macro language is not just for screen popping, because there are a wide number of macro commands that can be used for any sort of desktop automation. However, CallViewer does come with a large number of macro commands and variables related to calls, meaning that it can be quite useful to create user actions that manipulate calls.

In version 4, CallViewer has built-in actions that can be easily assigned to a button, hot key, or rule. These built-in actions allow a user to easily configure common call control features, such as making a call, or transferring a call. Whereas in earlier versions of CallViewer the user would have needed to write a custom macro to do any call control, from version 4 onwards the user should only need to create a user action to perform call control if they want to perform multiple call control commands, or wait for user input before performing a given command.

6.7.1 Which Extension?

Which Extension?

One facility that the macro commands provide that is not available using built-in actions in CallViewer is the ability to perform a call control command at a given extension. Almost all of the call control macro commands take an extension as the first parameter. If the first parameter is blank, the call control command will operate on the extension with which CallViewer is associated, otherwise it will use the extension entered in the macro command.

The following example will place an outbound call at the default extension to " 14809619000 ."

```
CallMake("", "01293608200", 1)
```

In contrast, the next example will place an outbound call at extension 400 to " 14809619000 ," regardless of whether this CallViewer is associated with extension 400 or not.

```
CallMake("400", " 14809619000 ", 1)
```

If the given extension is not in a position to make a call, both forms of the command would return an error and not continue.

6.7.2 Which Call?

Which Call?

Several of the macro commands that provide call control take a call index as a parameter. The call index is the 1-based index of the calls that are active at the given extension. For example, if a call alerts at an extension and is answered, before a second call alerts at the extension while queuing, the answered call is call index 1, and the alerting call is index 2. When the answered call is terminated, the alerting call becomes the only call and so has an index of 1.

When performing call control commands at the extension that is associated with CallViewer , you can also use an index of “0,” which causes the macro language to use the most suitable call at the extension.

For example, the following macro command will place the second call at this extension on hold:

```
CallHoldExclusive("", 2)
```

In the case of some macro commands, the command will find the most appropriate call index anyway. MiCC Office Server will also perform similar checks, which is useful when performing call control at another extension.

6.7.3 Blocking

Blocking

Macro commands are processed sequentially; the next macro command in the script will not be processed until the current command is complete. Whereas most macro commands take affect locally, e.g., the **SendKeys** macro command emulates keystrokes on the local computer, call control commands need to be passed to the , and ultimately on to the telephone system. At the same time though, wants to display an error if the macro command is not correct.

The result of this is that a call control macro command will block the current action's execution until it has received a response from as to whether the command was successful or not. returns a failure response if it believes that the given macro command is not applicable at the current time, e.g., requesting to answer a call when there are no calls at the given device. If returns success, it means that the command has been passed to the telephone system for execution. This does not mean that the related telephone system action has occurred yet however.

The following diagram indicates what information is passed between the three systems, and in an approximate order.

All of these events will occur very quickly, but comparatively, the time from receiving the call control request at the telephone system and it being processed will potentially be the longest. This means that can be processing a subsequent macro command before the telephone system has finished processing the first request.

Under normal circumstances this should not be a problem, however the following sample does give one example of when it would.

```
CallMake("", "62514", 0)
CallAnswer("62514", 1)
```

The first line places a call from the default extension to an internal device, 62514. The second line answers the first call at device 62514. Although macro execution will block until a response is received from the that the first command has been successfully passed to the telephone system, the second command will be processed before the internal call to "62514" is made. This means that the second command to answer a call at "62514" will fail, as no such call will exist at that moment in time.

Because monitors calls at only one extension, there are only two options for fixing this problem:

- Use a **Wait** macro command to put a delay between the two macro commands. A delay of between 1 and 1.5 seconds would be about right.

- Alternatively, alter the first line to make device 62514 call your extension, and then move the **CallAnswer** macro command into a rule that fires when an internal call alerts your extension from device 62514. This second option would not necessarily apply to any situation.

6.7.4 Call Control Example

Call Control Example

The following example prompts the user for an account code, which, having been entered is then set against the active call, and then the call is ended.

```
' Ensure that we have at least one call at our device
GotoIfNoCalls(“GotACall”, 0, 1)
' There are no calls at all, so end
End
GotACall:
' Prompt the user for an account code to set against this
' call and store the result in [Data1]
InputBox(“Enter an account code”, “ CallViewer ”, “”, 1)
' If no account code was entered, end the macro
GotoIfStrLen(“GotAcCode”, [Data1], 0, 1)
' Account code length was 0, so quit
End
GotAcCode:
' We have a valid account code - set it against the call
' via the telephone system
SetAccountCode(“”, 0, [Data1], 1)
' Rather than guess when the account code is set,
' we'll sit here and wait for it to update
DataSetNum(2, 20)
WaitForAcCodeLoop:
' Pause briefly
Wait(100)
' If the current Account Code = the Account Code user
' entered, then exit loop
GotoIfStrValue(“AcCodeSet”, [AccountCode], [Data1], 0)
' Decrement Data2 variable so we don't spin here
' for too long
DataSetNum(2, [Data2] - 1)
' See if Data2 has reached 0 - if it has, we've timed out,
' and not set the A/C Code yet
GotoIfNumValue(“WaitForAcCodeLoop”, [Data2], 0, 1)
' If we reach here it means that Data2 has reached 0
MessageBox(“Failed to set the account code”, 48, “ CallViewer ”)
End
AcCodeSet:
' OK, we must have set the account code by now,
' so now we can end the call
CallDrop(“”, 0)
```

An interesting area of the example is the **`WaitForAcCodeLoop`** section. This loop waits 2 seconds for the account code to change before allowing the code to continue. The loop starts by waiting for 100ms. It then checks the current account code,

and if it’s the account code entered by the user, will exit the loop. The [Data2] variable is used to limit the maximum number of times the code will wait for the account code to change. It starts at 20, and is decremented each time by 1. If it has not reached 0, the loop returns to the top to wait for another 100ms, otherwise the code displays a message to say that it has timed out waiting for the account code to change, and ends.

6.8 Advanced Topics

Advanced Topics

The following sections are intended for experienced users who want to further customize their CallViewer macros.

6.8.1 Multitasking

Multitasking

Because Windows is a multitasking operating system, it will perform some tasks at the same time that user actions are running. If your actions contain commands that invoke a task that will be performed in this way, you need to write the user actions to ensure that the requested multitasking operations have been fully completed before the next part of your user action executes.

Under Windows 98/ME, the use of the **YieldToOS** macro command after a macro statement that requests a multitasking operation is usually sufficient.

Under Windows 2000, or XP, you may have to write your action to use line labels and conditional branching (the “Goto” macro statements) to test whether multitasking operations have been completed. A simpler solution can be to use the **Wait** macro command to delay the action for a short period of time; however, this solution is not guaranteed to work on every installation due to timing differences. For example a 50ms wait on the test computer may be sufficient, but on other computers a 100ms wait may be required.

The following are operations that Windows performs in a multitasking fashion:

- Creating a window, whether this is the main application window, or a dialog.
- Starting applications (e.g., with the Shell macro command).
- Activating another window or application.
- Repainting the screen.
- Restoring, , or windows.

Note: Other features of an application could be performed in a multitasking manner, so if in doubt it is best to use **YieldToOs** anyway.

Also: If you send keystrokes to an application using **SendKeys** or **SendKeysEx**, you should call **YieldToOs** afterwards if the keystrokes would cause the application to perform one of the specified multitasking operations.

In earlier versions of it was not possible to have multiple macros running at the same time. One macro could call another macro, such that multiple macros could be active, but the calling macro would wait for the called macro to complete before continuing. In version 4 of this is no longer the case, and all user actions operate independently. This has certain ramifications:

The [**Data1**] to [**Data11**] macro variables are independent for each user action. If two actions are running concurrently then each action will have its own values for these variables. Even if the same action runs twice at the same time, the variables will still have their own values.

The [**Data1**] to [**Data11**] macro variables are always set to empty when an action instance starts, so there is no way of preserving data between actions being run using these macro variables.

If you need to share information between instances of running actions, use the **GlobalData*XXX*** set of macro commands. These allow you to read and write to named properties that last for the duration that is running, and are shared between all actions. This means that if action A sets a global property to “One,” and action B then sets the property to “Two,” both actions will read the property as “Two.” Global properties are also protected so that if two actions were running simultaneously, access to the property is automatically locked, so that two actions cannot try to simultaneously update the same property.

6.8.2 CallViewer as DDE Server

CallViewer as DDE Server

CallViewer can act as a DDE Server, as well as initiating DDE conversations with other DDE Servers. Because CallViewer acts as a DDE Server, other applications can initiate DDE conversations with CallViewer to send it commands, and request information about CallViewer .

CallViewer 's Application Name is "CALLVIEW," supporting a single topic called "System."

Because fewer modern applications provide a means of acting as a DDE Client, you may find it easier to drive CallViewer using the CallViewer Link Control via VBScript.

Requesting Data From CallViewer

After a conversation with CallViewer has been started you can request the value of any CallViewer w macro variable, as follows:

1. Set the "DataType" item to be the name of the macro variable you want to request, e.g., to request the [**Digits**] macro variable, you would 'poke' the "DataType" item with the value "Digits".

2. Request the "Data" item. This is automatically set to the contents of the macro variable that "DataType" is set to, when the "DataType" item is set.

Sending Commands To CallViewer

A command is an instruction sent to an application in a DDE conversation. The command could be an instruction for the application to perform an action such as opening a specified file.

When you use CallViewer as a DDE server, CallViewer accepts any valid native CallViewer macro command or even an entire macro script as a DDE command. If you send an entire macro script to CallViewer using a DDE conversation, then each line of the macro script must be separated by line feed (ANSI character 10) followed by carriage return (ANSI character 13).

After submitting a command, you can request the DDE items "ErrStr" or "ErrVal" to check the result.

- "ErrStr" returns the textual description of the error that occurred for the last submitted command.

- "ErrVal" returns the error number of the error that occurred for the last submitted command. A value of "0" indicates that the last command was successful. If the number is non-zero, then "ErrStr" can be requested to get a textual description of the same error code.

See the Macro Commands topics in the Reference section for a list of macro commands that could be sent to CallViewer .

DDE Server Examples

The following are examples of using CallViewer as a DDE server from Microsoft Access using the Access Basic programming language.

These are not examples of CallViewer 's own macro language.

Requesting The Value Of A Macro Variable

```
Dim lChannel As Long
' Initiate DDE conversation with Callview.
lChannel = DDEInitiate("Callview", "System")
' Set DataType item to be "Calls"
' Macro Variable Reference.
```

```
DDEPoke lChannel, "DataType", "Calls"
' Request it's value and show it in a
' Messagebox window.
MsgBox DDERequest(lChannel, "Data")
' Terminate all DDE conversations.
DDETerminateAll
```

Sending A Single Macro Command

```
Dim lChannel As Long
' Initiate DDE conversation with Callview.
lChannel = DDEInitiate("Callview", "System")
' Turn error handling off.
On Error Resume Next
' Send CallMake() command to Callview.
DDEExecute lChannel, "CallMake(""",""400"", 1)"
' If the command failed, then
' request the error values.
If Err <> 0 Then
Dim sErrVal As String
Dim sErrStr As String
sErrVal = DDERequest(lChannel, "ErrVal")
sErrStr = DDERequest(lChannel, "ErrStr")
' Show error string in message window.
MsgBox sErrStr
End If
' Terminate all DDE conversations.
DDETerminateAll
```

Sending An Entire Macro Script

```
Dim lChannel As Long
' Initiate DDE conversation with Callview.
lChannel = DDEInitiate("Callview", "System")
' Build a 3 line macro script into
' a string variable.
Dim s As String
s = ""
s = s & "CallMake(""",""400"", 1)" & Chr(10) & Chr(13)
s = s & "Wait(2500)" & Chr(10) & Chr(13)
s = s & "CallDialDigits(""", 0, ""#200"")" & Chr(10) & Chr(13)
' Turn error handling off.
On Error Resume Next
' Send macro script to Callview.
DDEExecute lChannel, s
' If the command failed, then
' request the error values.
```

```
If Not (0 = Err) Then
Dim sErrVal As String
Dim sErrStr As String
sErrVal = DDERequest(lChannel, "ErrVal")
sErrStr = DDERequest(lChannel, "ErrStr")
' Show error string in message window.
MsgBox sErrStr
End If
' Terminate all DDE conversations.
DDETerminateAll
```

7 Link Control

CallViewer Link Control

The Callview Link control is an Active X control that can be used in Active X aware development environments to integrate a custom application with a running instance of CallViewer . It can also be used in a VBScript user action that runs within CallViewer .

The control provides various methods, properties and events to integrate an application with CallViewer . Some examples of what you can achieve are as follows:

- Perform call control operations such as making a call, or changing the agent state of an agent.
- Find out information about the currently selected call, such as the telephone number, DID number, etc.
- Be notified of events occurring such as a new call alerting the associated CallViewer extension, or the forward state of the device changing.

The Link control communicates with a running instance of CallViewer on the user's desktop, and is therefore limited to information and events related to the extension that CallViewer is associated with. Call control operations can be performed on any device in the system, for example instructing CallViewer to transfer a call at another extension to voice mail.

Note: The Link Control topics explain how to use it and cover the properties, methods, and events that are supported. It does not explain how to program in VBScript.

8 Adding the Control to your Application

Adding the Control to your Application

The control can be used with development environments that interface with ActiveX, such as Microsoft Visual Basic or Microsoft Access. It can also be used in CallViewer VBScript macros.

Note: The Callview Link control can be used in Callviewer and Connection Assistant .

Microsoft Access

The control can be used with Microsoft Access 97 and above. A typical example of use would be to add the control to the form where contact information is displayed, and then use events from the control to automatically locate the correct record in the form when a call is answered at the user's extension.

To add the control to your form, perform the following actions. The instructions here refer to Microsoft Access 2000, but the principals will be the same for other versions of Access.

1. Open the relevant form in Design mode.

2. Click the **More Controls** button on the toolbox, and then choose **Callview Link Control** from the menu. If you cannot find the entry in the menu, CallViewer might not be installed on your development computer.

3. Drag on your form where you want the control to be located. The control is invisible when your form is in "View" mode for data entry, so the size and location of the control is irrelevant.

4. Name the control using the Properties window. Examples in this help assume that the control is given a name of "axCallview".

The control is now ready to use. The properties of the control are disabled when designing the form since they are only of use when running the form since they represent information about the state of CallViewer or the current calls at the user's extension. Enter the Visual Basic editor in Microsoft Access and write the necessary code to perform the operations that you require.

Microsoft Visual Basic

The control can be used with Microsoft Visual Basic 5 and above. A typical example of use would be to perform special routing rules on calls that could not normally be performed with Intelligent Router , or to integrate CallViewer with a custom application used by a company.

To add the control to your Visual Basic application, perform the following actions. The instructions here refer to Microsoft Visual Basic 6, but the principals will be the same for other versions of Visual Basic.

1. Open the relevant form in Form mode.

2. From the Tools menu, choose **Controls**. In the dialog that appears, locate the Callview Link Control in the list. Place a check mark next to the name, and then click **OK**.

   The Callview Link Control is now available in your toolbox.

3. Select the control, and then drag on your form where you want the control to be located. The control is invisible at runtime, so the size and location of the control is irrelevant.

4. Name the control using the Properties window. Examples in this help assume that the control is given a name of "axCallview".

The control is now ready to use.

Using VBScript Inside CallViewer

CallViewer has the ability to run VBScript user actions created using the Macro Editor. In such a scenario, CallViewer instantiates the Link control for use within the user action, allowing the control to be referenced using the Callview name.

For example, within a CallViewer VBScript macro, the user can make a call using the command:

```
Callview.CallMake "", " 14809619000 ", True
```

To create a VBScript macro in CallViewer :

1. Right-click the CallViewer traybar icon on the desktop, and choose Actions from the menu.
2. In the Action Manager click **Add** and select the **User Macro** option from the bottom of the menu.
3. In the macro editor, choose VBScript from the drop-down menu on the right of the toolbar.

You can now write the VBScript macro in the macro editor, and use the Link control methods and properties to interface with CallViewer .

Note: You cannot use the Callview Link control events in a VBScript macro. However, the events directly relate to rule types in the Rule Manager, and so it is recommended that a rule is used to fire off a macro, rather than use events in the macro itself.

Tip: When in the macro editor, the Insert button on the toolbar displays a list of all Callview Link control methods and properties, making it very easy to find the correct command and syntax that is required.

CallViewer has the ability to run VBScript user actions created using the Macro Editor. In such a scenario, CallViewer instantiates the Link control for use within the user action, allowing the control to be referenced using the Callview name.

For example, within a CallViewer VBScript macro, the user can make a call using the command:

```
Callview.CallMake "", " 14809619000 ", True
```

To create a VBScript macro in CallViewer :

1. Right-click the CallViewer traybar icon on the desktop, and choose Actions from the menu.
2. In the Action Manager click **Add** and select the **User Macro** option from the bottom of the menu.
3. In the macro editor, choose VBScript from the drop-down menu on the right of the toolbar.

You can now write the VBScript macro in the macro editor, and use the Link control methods and properties to interface with CallViewer .

Note: You cannot use the Callview Link control events in a VBScript macro. However, the events directly relate to rule types in the Rule Manager, and so it is recommended that a rule is used to fire off a macro, rather than use events in the macro itself.

Tip: When in the macro editor, the Insert button on the toolbar displays a list of all Callview Link control methods and properties, making it very easy to find the correct command and syntax that is required.

Using VBScript Outside of CallViewer

You can also use VBScript with the Link control when calling from another VBScript-enabled application, e.g., CSCRIPT.EXE, Microsoft Office, etc. Unlike when using VBScript from within CallViewer , you will need to instantiate your own instance of the control. The following code sample shows you how to instantiate the control yourself:

```
Dim objCallview
```

```
Set objCallView = CreateObject("Callview.Link.1")
```

You could then reference the object via the "objCallview" variable, e.g.:

```
objCallview.CallMake "", " 14809619000 ", False
```

When you are finished you must de- initialize the object by setting it to Nothing as follows:

```
Set objCallview = Nothing
```

Note: You cannot use the Callview Link control events in a VBScript macro. Furthermore, when using VBScript outside of CallViewer , you cannot use the Rule Manager's rules to act as a workaround for the lack of events.

Also: If you find that you need events you should either use VBScript macros within CallViewer itself, or consider using a full-blown programming environment to host the Callview Link control instead.

You can also use VBScript with the Link control when calling from another VBScript-enabled application, e.g., CSCRIPT.EXE, Microsoft Office, etc. Unlike when using VBScript from within CallViewer , you will need to instantiate your own instance of the control. The following code sample shows you how to instantiate the control yourself:

```
Dim objCallview
Set objCallView = CreateObject("Callview.Link.1")
```

You could then reference the object via the "objCallview" variable, e.g.:

```
objCallview.CallMake "", " 14809619000 ", False
```

When you are finished you must de- initialize the object by setting it to Nothing as follows:

```
Set objCallview = Nothing
```

Note: You cannot use the Callview Link control events in a VBScript macro. Furthermore, when using VBScript outside of CallViewer , you cannot use the Rule Manager's rules to act as a workaround for the lack of events.

Also: If you find that you need events you should either use VBScript macros within CallViewer itself, or consider using a full-blown programming environment to host the Callview Link control instead.

9 Using the Control Introduction

Using the Control

This section provides information on how to get the best out of the control. Using the control in the manner described in this section will optimize how your application communicates with CallViewer .

10 Using Methods

Methods

The vast majority of the methods provided by the Callview Link control are the same as the methods provided by the Macro Language, e.g., both the macro language and the Link control have **AppActivateLike** commands and **SendKeys** commands. This means that examples in can often be easily converted to VBScript.

For example, a macro that activated Notepad and typed the current call's telephone number would look like:

```
AppActivateLike "Untitled - Notepad"
YieldToOs
SendKeys "The phone number is " + [Digits]
A similar macro in VBScript, would look like the following (it is assumed that the
macro has been written inside ):
Callview.AppActivateLike "Untitled - Notepad"
Callview.DoCommand "YieldToOs"
Callview.SendKeys "The phone number is " + Callview.Digits
```

Although the Link control does not have a **YieldToOs** command itself, it can use the macro language's command using the **DoCommand** method.

There are some commands that are not necessary in the Link control, because VBScript has a means to do the command already. An example of this would be any of the **Goto*XXX*** commands, which jump conditionally based on the outcome of a comparison.

The following Macro excerpt checks the telephone number of the caller, and branches if the dial code is a particular value:

```
GotoIfStrValueLeft("LocalNumber", [Digits], "01293", 5, 0)
' Telephone number not local, end the macro
End
LocalNumber:
' Telephone number is local - output message
  MsgBox "Local Number"
In VBScript, this would become:
If Left(Callview.Digits, 5) = "01293" Then
' Telephone number is local - output message
  MsgBox "Local Number"
Else
' Telephone number not local
' Do something else
End If
```

Information on all the methods provided by the Callview Link control can be found in the Reference section, Link Control Methods topics .

11 Using Properties

Using Properties

Similarly to methods, the Callview Link control's properties are invariably identical to the variables available in the Macro language, for example the [**CTIServerName**] macro variable is identical to the **CTIServerName** property.

However, it is very important to remember that when using the Link control, it is usually communicating from a different process than the one that is running in, e.g., the Link control is being hosted by Microsoft Access, and so is a different process. This means that requesting a property does not provide an immediate response. The busier that the computer and/or is, the longer it will take for the property to be returned to the calling application.

This means that it is better to request a property once, and cache the result, then to keep on asking for the same property, unless you know it has changed. For example:

```
Dim szNumber
' Store a copy of the current phone number
  szNumber = Callview.Digits
' Display it
  MsgBox szNumber
' Select a new call
  Callview.CallSelect 3
' Display the phone number of call 3
  MsgBox szNumber
```

The script does not do what it says it will do. Although the phone number has been cached in 'szNumber,' the last line is meant to display the phone number for the third call, yet the phone number has not been recached. After the call to **CallSelect**, the result of the Digits property will probably be different, so the information should have been cached again:

```
' Select a new call
Callview.CallSelect 3
szNumber = Callview.Digits
' Display the phone number of call 3
MsgBox szNumber
```

Information on all properties of the Link control can be found in the Reference section, Link Control Properties topics.

12 Using Events

Using Events

When you host the Callview Link control in another application, e.g., Microsoft Access, Visual Basic, etc., you can be notified of events occurring in . Events are telephony based, i.e., they inform the control when the call or extension status changes, for the extension that is associated with. So, for example, there are events to depict when a call starts alerting, when it is answered, when it is held, and so on.

Each event provides several pieces of information in relation to what actually happened. For example call related events will provide information on the call, including the telephone number, , etc.

Further information on events can be found in the Reference section, Link Control Events topics .

Note:To receive call events from the Link control, the Initialise method must be called to tell the control that the application would like to receive Link control events. It is not necessary to call the **Uninitialise** method when your application terminates, although it is good programming .

Using The Information Provided

It is recommended that you use the information passed in the event, rather than obtaining the same information via the control's properties. There are two reasons for this:

There could be more than one call at the associated extension, and the ActiveX control will return a call property for the currently selected call within the call list if referenced by a property rather than using the event parameters. From an external application's point of view this would be an arbitrary call, rather than the call that caused the event to fire. For example, for account code capture, it is recommend that one uses the **ExtAccountCodeEntered** event procedure, as opposed to specifically querying the **AccountCode** ActiveX control property.

The information against the event has already been provided. If you choose to query the control for further information, then you have to wait for the information to be returned.

Avoiding Blocking

It's important to remember that when an event fires, is calling your code. This means that spending a long time doing something inside an event is likely to delay 's ability to process messages and so on. You could even potentially lock .

For example, say you wrote an application that processed the **CallNew** event, which fires when a call starts alerting your extension. If you display a message box of some form in that event, the event cannot return execution to the calling application until the message box is closed. This is going to delay other events from arriving at your application.

does not have this problem when it fires rules, because it allocates each rule its own thread when it needs to process an action.

Call

In a complex application of events it is recommended that events are used to build a local copy of the calls, rather than polling the control for changes to call properties, since polling does not result in am immediate return of data requested.

A complex application would be one that needed to rely on matching up several events, and/or looking at several calls at once. If you just needed to process one or two events and then perform a short script based on the information in the event itself, building a local copy of the calls is overkill; using the information from the event works much better.

Note: When building a local copy of the calls, each call needs to be indexed by the distant end device (the device that the call is connected to).

13 Reference Introduction

Reference Introduction

This section provides reference information for the CallViewer Macro Language and the CallViewer Link Control.

14 Macro Commands Introduction

Macro Commands Introduction

The CallViewer macro language contains commands and variables, which are described in the following sections.
The Macro Commands section documents the commands that are available in the CallViewer Macro Language.

14.1 ActivateApp

ActivateApp

This method restores and activates the application window that has the titlebar text matching the string specified in the parameters. The match is performed using wildcards. The command also changes the focus to the named application window and restores if if .

If a matching application is not found an error occurs.

Syntax:

ActivateApp(WindowTitle)

Parameter:

WindowTitle: The title of the application to activate. This string can contain wildcards, where an * represents 0 or more characters, and a ? represents one character. For example, the Microsoft Outlook titlebar text could be matched with either "Microsoft Out*", "*Outlook", "*soft Out", "??????? Outlook", etc.

Example:

```
' Set the focus to Notepad.
ActivateApp("* - Notepad")
YieldToOS
```

Notes:

- If there is more than one instance of the given application, the operating environment arbitrarily selects the one to activate.
- The command will cause an error if the window exists but is hidden. There are many application windows that are always open under Windows but are not visible. Examples are the NetDDE application window ("NetDDE") and the clipboard server application ("ClipSrv").
- This command replaces **AppActivateLike** and **AppActivateLikeRight** in version 4.1. These commands still exist, though they may be deprecated in the future.

14.2 ActivateChild

ActivateChild

This method restores and activates the child window of an application. Both the application and the child window are identified by specifying the titlebar text in the command's parameters. Wildcards can be used in the parameters The command also changes the focus to the named child window and restores it if .

If a matching application is not found an error occurs.

Syntax:

ActivateChild(WindowTitle, ChildWindowTitle)

Parametesr:

- **WindowTitle**: The title of the application to activate. This string can contain wildcards, where an * represents 0 or more characters, and a ? represents one character. For example, the Microsoft Outlook titlebar text could be matched with either "Microsoft Out*", "*Outlook", "*soft Out", "??????? Outlook", etc.

- ChildWindowTitle: The title of the child window to activate. This string can contain wildcards, similar to the WindowTitle parameter.

Example:

```
' Restore and activate a document in Microsoft Word.
ActivateChild("*- Microsoft Word","document1*")
YielToOS
```

Notes:

- If there is more than one instance of the application or child window, the operating environment arbitrarily selects the one to activate.

- The command will cause an error if the application window exists but is hidden. There are many application windows that are always open under Windows but are not visible. Examples are the NetDDE application window ("NetDDE") and the clipboard server application ("ClipSrv").

- The command does not set the focus to the specified application window.

- This command replaces **AppActivateLikeChild** and **AppActivateLikeRightChild** in version 4.1. Although these other commands still exist, they may be deprecated in the future.

14.3 ActiveXScriptRun

ActiveXScriptRun

Runs an ActiveX script. The script can be a single line of an ActiveX script such as VBScript, or several lines together, each separated by a carriage return and line feed.

Syntax:

ActiveXScriptRun(ScriptHostName, Script)

Parameters:

- **ScriptHostName**: The name of the ActiveX scripting engine that the script is written in, e.g., “VBScript.”
- **Script**: The script to execute. The script must still conform to the rules of the script engine. If it does not, the command will return an error.

Example:

```
' Show a message box window using VBScript.
ActiveXScriptRun(“VBScript”, “MsgBox ”“Test”“, 0, ”“ CallViewer ”“”)
```

Notes:

You can write user actions in any ActiveX script, such as VBScript, without the need to use this method within the CallViewer macro language. This can provide better performance, and may be easier if the length of the ActiveX scripts get reasonably long.

14.4 AppActivateLastFoc

AppActivateLastFoc

Restores and activates the last application window to have the focus, other than CallViewer . The **AppActivateLastFoc** command changes the focus to the application window and restores it if minimized .

Syntax:

AppActivateLastFoc(IgnorePopupWindows)

Parameter:

I**gnorePopupWindows**: This parameter controls whether the command only considers application windows when deciding which window was last active. Valid settings are as follows:

Value	Description
0	Restores and activates the last window to have the focus, regardless of what type of window it is. Thi option is not recommended.
1	Restores and activates the last application window to have the focus that meets one of the following conditions: • The application window is modal and has a double border that may have been created with or without a title bar. • The application window is not a popup window unless it is a modal dialog, as defined by its window style.

Example:

```
' Activate the last application
' window to have the focus.
AppActivateLastFoc(1)
YieldToOS
```

14.5 AppActivateLastFocCopyText

AppActivateLastFocCopyText

Restores and activates the last application window (other than CallViewer) to have the focus and copies the text that is selected within it to the clipboard. The **AppActivateLastFocCopyText** command changes the focus to the application window and restores it if minimized .

The method that is used to copy the text from the application is the one specified in the Go Dial Action setting on the Call Control tab of the Options dialog.

Syntax:

AppActivateLastFocCopyText(IgnorePopupWindows)

Parameter:

IgnorePopupWindows: This parameter controls whether the command only considers application windows when deciding which window was last active. Valid settings are as follows:

Value	Description
0	Restores and activates the last window to have the focus, regardless of what type of window it is. Thi option is not recommended.
1	Restores and activates the last application window to have the focus that meets one of the following conditions: • The application window is modal and has a double border that may have been created with or without a title bar. • The application window is not a popup window unless it is a modal dialog, as defined by its window style.

Example:

```
' Activate the last application window to have
' the focus and copy the text that is selected.
AppActivateLastFocCopyText(1)
YieldToOS
```

Notes:

If you use this command in a user action, and then assign that action as the Go Dial Action, you will create an infinite loop when you try to Go Dial.

14.6 AppActivateLike

AppActivateLike

This method restores and activates the application window that has the left part of its titlebar text matching the string specified in the parameters. The command also changes the focus to the named application window and restores it if .

If a matching application is not found an error occurs.

Note: Starting with version 4.1, this command has been replaced with ActivateApp. This command may be deprecated in future versions of the macro language.

Syntax:

AppActivateLike(WindowTitle)

Parameter:

WindowTitle: The left part of the title of the window to activate. The name that appears in the titlebar need not be fully specified. For instance “Calculat” would still activate an open application with titlebar text “Calculator”. The comparison is also not case sensitive; i.e., “Calculator” and “calculator” appear identical.

Example:

```
' Set the focus to Notepad.
AppActivateLike(“Untitled - Notepad”)
YieldToOS
```

Notes:

- If there is more than one instance of the given application, the operating environment arbitrarily selects the one to activate.
- The command will cause an error if the window exists but is hidden. There are many application windows that are always open under Windows but are not visible. Examples are the NetDDE application window (“NetDDE”) and the clipboard server application (“ClipSrv”).
- It may be more preferable to use the **AppActivateLikeRight** command instead of **AppActivateLike** in some circumstances. Unlike **AppActivateLike**, the **AppActivateLikeRight** command can identify an application by matching the right portion of its titlebar text instead of the left part.

14.7 AppActivateLikeChild

AppActivateLikeChild

Restores and activates the child window of an application. Both the application and the child window are identified by specifying the leftmost part of their titlebar text in the command's parameters. The command also changes the focus to the named child window and restores it if minimized .

If a matching application is not found an error occurs.

Note: Starting with version 4.1, this command has been replaced with ActivateChild. This command may be deprecated in future versions of the macro language.

Syntax:

AppActivateLikeChild(WindowTitle, ChildWindowTitle)

Parameters:

- **WindowTitle**: The left part of the title of the application window to activate.

  The name that appears in the titlebar need not be fully specified. For instance, "Calculat" would still activate an open application with titlebar text "Calculator". The comparison is also not case sensitive; i.e., "Calculator" and "calculator" appear identical.

- **ChildWindowTitle**: The left part of the title of the child window within the given application window.

  The name that appears in the titlebar need not be fully specified. For instance, "Exam" would still activate a child window with titlebar text "Example." The comparison is also case insensitive.

Example:

```
' Restore and activate a document in Microsoft Word.
AppActivateLikeChild("document 1 - Microsoft Word","document1")
YieldToOS
```

Notes:

- If there is more than one instance of the application or child window, the operating environment arbitrarily selects the one to activate.

- The command will cause an error if the application window exists but is hidden. There are many application windows that are always open under Windows but are not visible. Examples are the NetDDE application window ("NetDDE") and the clipboard server application ("ClipSrv").

- The command does not set the focus to the specified application window.

14.8 AppActivateLikeRight

AppActivateLikeRight

Restores and activates the application window that has the right side of the titlebar text matching the string specified in the parameters. The command also changes the focus to the named application window and restores it if minimized .

If a matching application is not found an error occurs.

Note: Starting with version 4.1, this command has been replaced with ActivateApp. This command may be deprecated in future versions of the macro language.

Syntax:

AppActivateLikeRight(WindowTitle)

Parameter:

WindowTitle: The rightmost part of the title of the application window to activate.

The name that appears in the titlebar need not be fully specified. For instance, “culator” would still activate an open application with titlebar text “Calculator”. The comparison is also not case sensitive; i.e., “Calculator” and “calculator” appear identical.

Example:

```
' Set the focus to Notepad by matching the right part
' of the titlebar text since Notepad's titlebar shows
' "Untitled - Notepad".
AppActivateLikeRight("Notepad")
YieldToOS
```

Notes:

- If there is more than one instance of the application, the operating environment arbitrarily selects the one to activate.
- The command will cause an error if the window exists but is hidden. There are many application windows that are always open under Windows but are not visible. Examples are the NetDDE application window (“NetDDE”) and the clipboard server application (“ClipSrv”).

14.9 AppActivateLikeRightChild

AppActivateLikeRightChild

Restores and activates the child window of an application that has the left part of its titlebar text matching the string specified in “the parameters. The application is identified by specifying the right part of its titlebar text in the parameters. The command also changes the focus to the named child window and restores it if .

If a matching application is not found an error occurs.

Note: Starting with version 4.1, this command has been replaced with ActivateChild. This command may be deprecated in future versions of the macro language.

Syntax:

AppActivateLikeRightChild(windowtitle, childwindowtitle)

Parameters:

- **WindowTitle**: The rightmost part of the title of the application window to activate.

  The name that appears in the titlebar need not be fully specified. For instance, “culator” would still activate an open application with titlebar text “Calculator”. The comparison is also not case sensitive; i.e., “Calculator” and “calculator” appear identical.

- **ChildWindowTitle**: The left part of the title of the child window within the given application window.

  The name that appears in the titlebar need not be fully specified. For instance, “Exam” would still activate a child window with titlebar text “Example”. The comparison is also case insensitive.

Example:

```
' Restore and activate a document in Microsoft Word.
AppActivateLikeRightChild(“Word”,“Document1”)
YieldToOS
```

Notes:

- If there is more than one instance of the application or child window, the operating environment arbitrarily selects the one to activate.

- The command will cause an error if the specified application window exists but is hidden. There are many application windows that are always open under Windows but are not visible. Examples are the NetDDE application window (“NetDDE”) and the clipboard server application (“ClipSrv”).

- Unlike the **AppActivateLikeRight** command, the **AppActivateLikeRightChild** command does not set the focus to the specified application window.

14.10 AppActivateLikeShell

AppActivateLikeShell

Restores and activates the application window that has the left part of its titlebar text matching the string specified in the parameters. If no window can be found, then the command launches the filename specified in the parameters.

If the filename provided cannot be found, an error will occur.

Syntax:

AppActivateLikeShell(WindowTitle, Filename)

Parameters:

- **WindowTitl**e: The left part of the title of the application window to activate.

  The name that appears in the titlebar need not be fully specified. For instance, “Calc” would still activate an open application with titlebar text “Calculator”. The comparison is also not case sensitive; i.e., “Calculator” and “calculator” appear identical.

- **Filename**: The name of the filename to open if the given window cannot be found. Typically this will be an executable file.

  The filename provided will need to include the full pathname to the file location, unless the given file is in the list of paths that Windows searches for executable files. This typically includes the current working directory, the Windows folder, and the System or System32 folders.

Example:

```
' Set the focus to Notepad application if open already,
' but run NOTEPAD.EXE if the application is not open.
AppActivateLikeShell(“Untitled - Notepad”,“NOTEPAD.EXE”)
YieldToOS
```

Notes:

- If there is more than one instance of the application, the operating environment arbitrarily selects the one to activate.

- The command will cause an error if the window exists but is hidden. There are many application windows that are always open under Windows but are not visible. Examples are the NetDDE application window (“NetDDE”) and the clipboard server application (“ClipSrv”).

- It is wise to place the **YieldToOS** command after the **AppActivateLikeShell** command. Since running applications under Windows is a multitasking operation, a **YieldToOS** statement after these types of commands ensures that they fully complete before execution passes to the remaining part of the macro script.

14.11 AppCopyText

AppCopyText

Copies the text that is selected within the current active application to the clipboard. The method that is used to copy the text from the application is the one specified in the Go Dial Action setting on the Call Control tab of the Options dialog.

Syntax:

AppCopyText

Parameters:

None.

Example:

```
' Activate the Notepad application window and copy the
' text that is selected within the current document.
AppActivateLike("Untitled - Notepad")
YieldToOS
' Copy selected text.
AppCopyText
```

Notes:

- If you use this command in a user action, and then assign that action as the Go Dial Action, you will create an infinite loop when you try to Go Dial.
- You can use the **AppCopyTextEx** command instead of **AppCopyText** which copies the selected text in the active application but uses a method to copy the text that you specify.

14.12 AppCopyTextEx

AppCopyTextEx

Copies the text that is selected within the current active application to the clipboard using a specific method to copy the selected text.

Syntax:

AppCopyTextEx(CopyMethod, CustomKeystrokes)

Parameters:

- **CopyMethod**: A numerical value which specifies the method that is used to copy the text. It can be one of the following values:

Value	Method	Description
0	Send WM_COPY	Sends the Windows WM_COPY message to the control that has the inpu focus. The success of this method is dependent on the application being used.
1	Send Ctrl-C	Simulates pressing the Control and C keys in the active application. This the recommended method, as almost all applications support such a keystroke.
2	Send Ctrl-Insert	Simulates pressing the Control and Insert keys in the active application. will often work when the Send Ctrl-C option does not.
3	Custom keystrokes	Simulates pressing the keystrokes specified in the second parameter to t command.
4	Send WM_GETTEXT	Sends the Windows WM_GETTEXT message to the control that has the input focus. This gets all the text in the given control, but success is dependent on the application being used.

- **CustomKeystrokes**: This setting denotes the custom keystrokes to send to the active application when using option 3 in the "CopyMethod" parameter. The format of the keystrokes is defined in the **SendKeys** command.

Example:

```
' Activate the Notepad application window and copy the
' text that is selected within the current document.
AppActivateLike("Untitled - Notepad")
YieldToOS
' Copy selected text using the Ctrl-c method.
AppCopyTextEx(1, "")
```

14.13 AppWindowHide

AppWindowHide

Hides the given application window, identified by the leftmost part of the window's titlebar text.

If the window cannot be found, an error occurs.

Syntax:

AppWindowHide(WindowTitle)

Parameter:

WindowTitle: The left part of the title of the application window to hide.

The name that appears in the titlebar need not be fully specified. For instance, "Calc" would still activate an open application with titlebar text "Calculator". The comparison is also not case sensitive; i.e., "Calculator" and "calculator" appear identical.

Example:

```
' Hide the Notepad window.
AppWindowHide("Untitled - Notepad")
YieldToOS
```

Notes:

- If there is more than one instance of the application, then the operating environment arbitrarily selects one.
- You must use the **AppWindowHide** command with caution especially if you want to hide the window itself. If your action does not show the window again using the **AppWindowShow** command, you may have to restart Windows in order to use because it will have completely disappeared off your desktop.

14.14 AppWindowMode

AppWindowMode

Restores, minimizes , or maximizes the application window, identified by the leftmost part of the window's titlebar text.

Syntax:

AppWindowMode(WindowTitle, Mode)

Parameters:

- **WindowTitle**: The left part of the title of the application window to affect.

  The name that appears in the titlebar need not be fully specified. For instance, "Calc" would still activate an open application with titlebar text "Calculator". The comparison is also not case sensitive; i.e., "Calculator" and "calculator" appear identical.

- **Mode**: A numerical value that specifies what action to take on the given window, as follows:

Value	Action
0	Restore the window
1	Minimize the window
2	Maximize the window
3	Size the window to its normal size

Example:

```
' minimize Notepad.
AppWindowMode("Untitled - Notepad", 1)
YieldToOS
```

14.15 AppWindowMoveTo

AppWindowMoveTo

Moves the application window to a given position on the screen.

Syntax:

AppWindowMoveTo(WindowTitle, Xpos, Ypos)

Parameters:

- **WindowTitle**: The left part of the title of the application window to affect.
- The name that appears in the titlebar need not be fully specified. For instance, “Calc” would still activate an open application with titlebar text “Calculator”. The comparison is also not case sensitive; i.e., “Calculator” and “calculator” appear identical.
- **Xpos**: The horizontal position of the leftmost edge of the given window to move the window to. This is measured in pixels. A valid range for this value depends on the current screen resolution being used.
- **Ypos**: The vertical position of the topmost edge of the given window to move the window to. This is measured in pixels. A valid range for this value depends on the current screen resolution being used.

Example:

```
' Move the Notepad window to the co-ordinates
' 100, 100 on the desktop.
AppWindowMoveTo(“Untitled - Notepad”, 100, 100)
YieldToOS
```

Notes:

If you specify an Xpos that is greater than the screen width, or a Ypos that is greater than the screen height, you will move the given window off the screen altogether, which may make it difficult for the user to return the window to a visible portion of the screen.

14.16 AppWindowSetOrder

AppWindowSetOrder

Changes the order of a window relative to other windows. This is used to bring a particular window to the top of all windows, or move it behind other windows.

Syntax:

AppWindowSetOrder(WindowTitle, Order)

Parameters:

- **WindowTitle**: The left part of the title of the application window to change the order of.

  The name that appears in the titlebar need not be fully specified. For instance, “Calc” would still activate an open application with titlebar text “Calculator.” The comparison is also not case sensitive; i.e., “Calculator” and “calculator” appear identical.

- **Order**: A numerical value that depicts how the window order for the given window should change. It can be one of the following values:

Value	Action
0	Place the window above all other windows. Any windows that are defined as “always on top” will rer above this window.
1	Place the window behind all windows. If the window is set to always stay on top of all windows, ther setting is revoked.
-1	Place the window above all other windows, and sets it so that it will stay on top of all windows.
-2	Revokes the setting for this window to always stay on top of all windows, without affecting its positic the window order.

Example:

```
' Make the Notepad window “always on top”.
AppWindowSetOrder(“Untitled - Notepad”, -1)
YieldToOS
```

14.17 AppWindowShow

AppWindowShow

Shows the given application window, identified by the leftmost part of the window's titlebar text.

If the window cannot be found, then an error is generated.

Syntax:

AppWindowShow(WindowTitle)

Parameter:

WindowTitle: The left part of the title of the application window to show.

The name that appears in the titlebar need not be fully specified. For instance, "Calc" would still activate an open application with titlebar text "Calculator". The comparison is also not case sensitive; i.e., "Calculator" and "calculator" appear identical.

Example:

```
' Show the Notepad window.
AppWindowShow("Untitled - Notepad")
YieldToOS
```

Notes:

- This command would usually be used to "unhide" a window that had been hidden with **AppWindowHide**.
- If there is more than one instance of the application, the operating environment arbitrarily selects one.

14.18 Beep

Beep

Sounds a tone through the computer's speaker.

Syntax:

Beep

Parameters:

None.

Example:

Beep

Notes:

The frequency and duration of the beep depends on hardware, which may vary among different computers.

14.19 CallAnswer

CallAnswer

This command answers an alerting call at the given extension.

If there is no alerting call at the specified extension or the extension device is in a state that cannot facilitate answering a call, an error occurs.

Syntax:

CallAnswer(Extension, CallItem)

Parameters:

- **Extension**: The extension device to answer the call at. If a blank string is specified, then the extension assigned to the running instance of t will be used.
- **CallItem**: The index of the call to answer. Calls in the list are identified as 1 for the first call in the call list, 2 for the second call etc. A value of 0 will instruct to automatically answer the first alerting call that it finds against the given extension. In fact, if the call specified in this argument is not alerting the given extension, then the next alerting call is answered instead.

Example:

```
' Make a call from a specified extension to the current
' extension so that if the specified extension is
' in divert(forward), you can still immediately
' connect to it
'
' Obtain the distant extension from the user.
InputBox("Enter extension : ", "Connect To", "", 1)
' End macro if nothing was entered.
ExitMacroStrValue([Data1], "", 1)
' Make the call from the extension entered to your
' extension.
CallMake([Data1], [LocalExtension], 1)
Wait(300) ' Wait for the call to be set-up.
' Auto answer the call at your extension.
CallAnswer("", 0)
```

Notes:

- You can find out whether it is possible to answer a call at your extension by interrogating the value of the [**CanCallAnswer**] macro variable.
- You can generically find the device number of the extension associated with the current installation of by using the [**LocalExtension**] macro variable.

14.20 CallConference

CallConference

This command allows you to conference calls together at the given device, using the following rules:

If the calls at your extension are answered or already being conferenced, the **CallConference** command places the current calls on hold and prompts you to enter in the extension or telephone number of another party.

If there are held calls at your extension, the **CallConference** command joins all the calls together into a conference.

If the given extension device is in a state that cannot facilitate the conferencing of calls (or adding a new conference party), an error occurs.

Syntax:

CallConference(Extension)

Parameter:

Extension: The extension device to conference calls at. If a blank string is specified, then the extension assigned to the running instance of will be used.

Example:

```
' Places the current call on hold and invokes
' the Add Party window to dial a new number.
CallConference("")
' Wait for the new call to be created.
Wait(500)
' Join the new call and the existing call
' together in a conference.
CallConference("")
```

Notes:

You can find out whether it is possible to conference calls at your extension by interrogating the value of the [**CanCallConf**] macro variable.

You can generically find the device number of the extension associated with the current installation of by using the [**LocalExtension**] macro variable.

The maximum number of parties that may be joined together in a conference will depend on the telephone system that the is connected to. The maximum number of parties specified usually includes the device that you are calling from, thus the maximum number of calls you may conference together with yourself is often 1 less than the number actually specified. Refer to your telephone system documentation to obtain conferencing party limitations.

14.21 CallDialDigits

CallDialDigits

This command dials the specified digits over an existing call at the given extension.

If there is no answered call at the specified extension or the extension device is in a state that cannot facilitate digits, an error occurs.

Syntax:

CallDialDigits(Extension, CallItem, Digits)

Parameters:

- **Extension**: The extension device to dial the digits at. If a blank string is specified, the extension assigned to the running instance of will be used.

- **CallItem**: The index of the call to dial digits on. Calls in the list are identified as 1 for the first call in the call list, 2 for the second call etc. A value of 0 will instruct to automatically dial digits on the first answered call that it finds against the given extension.

  In fact, if the call specified is not in the answered state at the given extension, the next answered call is used instead.

Example:

```
' Make a call to the voice mail group.
CallMake("", "400", 1)
' Wait for the call to be answered.
Wait(2500)
' Enter the voice box of the local extension.
CallDialDigits("", 0, "#" + [LocalExtension])
' Obtain the password from the user.
InputBox("Enter password", "Voice Mail", "", 1)
' End macro if nothing was entered.
ExitMacroStrValue([Data1], "", 1)
' Dial the password to the voice mail.
CallDialDigits("", 0, [Data1])
```

Notes:

- You can find out whether it is possible to dial digits at your extension by interrogating the value of the [**CanCallDialDig**] macro variable.

- You can generically find the device number of the extension associated with the current installation of by using the [**LocalExtension**] macro variable.

14.22 CallDialDigitsInput

CallDialDigitsInput

This command displays an input box so that the user can enter in digits that they want to dial over an existing call. A default value can be specified which is initially shown in the input box.

If there is no answered call at the specified extension or the extension device is in a state that cannot facilitate dialing digits, an error occurs.

Syntax:

CallDialDigitsInput(Extension, CallItem, DefaultDigits)

Parameters:

- **Extension**: The extension device to dial the digits at. If a blank string is specified, the extension assigned to the running instance of CallViewer will be used.

- **CallItem**: The index of the call to dial digits on. Calls in the list are identified as 1 for the first call in the call list, 2 for the second call etc. A value of 0 will instruct CallViewer to automatically dial digits on the first answered call that it finds against the given extension.

In fact, if the call specified is not in the answered state at the given extension, the next answered call is used instead.

- **Digits** : The digits to dial on the specified call. Some characters have a special meaning:

Character	Meaning
!	This character can precede a feature code. By dialing features codes you can simulate extension feature being “accessed” on a station device. You can usually do this even w the [**CanCallDialDig**] macro variable returns a value that indicates that dialing digits o is unavailable. See your extension manual for a list of default feature code values.
P	Pause
F	Hookflash

- : The digits to dial on the specified call. Some characters have a special meaning depending on the telephone system that the Callview Gateway is connecting to.

14.23 CallDrop

CallDrop

This command ends a call at the given extension. The call must be an outbound external call, or answered.

If there is no external outbound or answered call at the specified extension, or the extension device is in a state that cannot facilitate dropping a call, an error occurs.

Syntax:

CallDrop(Extension, CallItem)

Parameters:

- **Extension**: The extension device to end the call at. If a blank string is specified, the extension assigned to the running instance of will be used.

- **CallItem**: The index of the call to end. Calls in the list are identified as 1 for the first call in the call list, 2 for the second call etc. A value of 0 will instruct to automatically drop the first call at the given extension that is outbound external or answered.

  In fact if the call specified at the given extension is not an outbound external call, or answered, the first such call at the extension is used instead.

Example:

```
' Drops the first call at your extension.
CallDrop("", 1)
```

Notes:

- You can find out whether it is possible to end a call at your extension by interrogating the value of the [**CanCallDrop**] macro variable.

- You can generically find the device number of the extension associated with the current installation of by using the [**LocalExtension**] macro variable.

14.24 CallDropAll

CallDropAll

This command ends all unheld calls at the given extension. It also places the device in an “on hook” state.

If the extension device is in a state that cannot facilitate ending calls, an error occurs.

Syntax:

CallDropAll(Extension)

Parameter:

Extension: The extension device to end all calls at. If a blank string is specified, then the extension assigned to the running instance of will be used.

Example:

```
' Drops all the calls at your extension.
CallDropAll(“”)
```

Notes:

- You can find out whether it is possible to end all calls at your extension by interrogating the value of the [**CanCallDropAll**] macro variable.
- You can generically find the device number of the extension associated with the current installation of by using the [**LocalExtension**] macro variable.

14.25 CallHoldExclusive

CallHoldExclusive

This command exclusively holds an external outbound or answered call at the given extension.

If there is no external outbound call or call in the answered state at the specified extension, or the extension device is in a state that cannot facilitate exclusively holding a call, then an error occurs.

Syntax:

CallHoldExclusive(Extension, CallItem)

Parameters:

- **Extension**: The extension device to exclusively hold the call at. If a blank string is specified, then the extension assigned to the running instance of will be used.

- **CallItem**: The index of the call to hold. Calls in the list are identified as 1 for the first call in the call list, 2 for the second call etc. A value of 0 will instruct to automatically hold the first call at the given extension that is outbound external or answered.

  In fact, if the call specified at the given extension is not an outbound external call, or answered, the next call at the extension is used instead.

Example:

```
' This macro performs the equivalent to the
' CallTransfer macro by placing the current
' call on hold and making a consultation call.
'
' Get the user to enter the party to call.
InputBox("Enter party : ", "Transfer", "", 1)
' End the macro if nothing was entered.
ExitMacroStrValue([Data1], "", 1)
' Exclusively hold the current call.
CallHoldExclusive("", 0)
' Wait for the call to be held
Wait(400)
' Make the call to the new party.
CallMake("", [Data1], 1)
```

Notes:

- You can find out whether it is possible to exclusively hold a call at your extension by interrogating the value of the [**CanCallHoldEx**] macro variable.

- You can generically find the device number of the extension associated with the current installation of by using the [**LocalExtension**] macro variable.

14.26 CallHoldSystem

CallHoldSystem

This command system holds (parks) an external outbound or answered call at the given extension.

If there is no external outbound call or call in the answered state at the given extension, or the extension device is in state that cannot facilitate placing a call on system hold, an error occurs.

Syntax:

CallHoldSystem(Extension, CallItem)

Parameters:

- **Extension**: The extension device to system hold (park) the call at. If a blank string is specified, then the extension assigned to the running instance of will be used.

- **CallItem**: The index of the call to hold. Calls in the list are identified as 1 for the first call in the call list, 2 for the second call etc. A value of 0 will instruct to automatically hold the first call at the given extension that is outbound external or answered.

  In fact, if the call specified at the given extension is not an outbound external call, or answered, the next call at the extension is used instead.

Example:

```
' System hold (park) the current call.
CallHoldSystem("", 0)
```

Notes:

- You can find out whether it is possible to system hold a call at your extension by interrogating the value of the [**CanCallHoldSys**] macro variable reference.

- You can generically find the device number of the extension associated with the current installation of by using the [**LocalExtension**] macro variable.

14.27 CallMake

CallMake

This command makes a new call at the given extension.

If the device is in a state that cannot facilitate making a call, an error occurs.

Syntax:

CallMake(Extension, DialString, AutoPrefix)

Parameters:

- **Extension**: The extension device to make a new call at. If a blank string is specified, the extension assigned to the running instance of CallViewer will be used.
- **DialString**: The telephone number to dial.
- **AutoPrefix**: This is a numerical argument that when set to “0” dials the digits exactly as entered in the “DialString” parameter.

When this value is set to “1”, the number to be dialed is affected by the dial rules configured within CallViewer . At a minimum this means that the outbound dial prefix will be included in the digits sent to the telephone system.

Example:

```
' Get the number to dial from the user.
InputBox(“Enter number :”,“Dial”,“”, 1)
' Make a call using the number.
CallMake(“”,[Data1], 1)
```

Notes:

The following characters within the “DialString” argument have a special meaning::

Character	Meaning
!	This character can precede a feature code. By dialing features codes you can simulate an extens feature being “accessed” on a station device. You can usually do this even when the [**CanCallDi**] macro variable returns a value that indicates that dialing digits on line is unavailable. See your extension manual for a list of default feature code values.
P	Pause
F	Hookflash

14.28 CallMakeAppActiveLast

CallMakeAppActiveLast

This command performs the same operation as the “Go Dial” action within . It “grabs” a telephone number or e-mail address from the last active application, and then either dials the telephone number, or creates a blank e-mail in the default e-mail application, addressed to the given e-mail address.

The command also changes the focus to the last active application window, and restores it if .

Syntax:

CallMakeAppActiveLast(IgnorePopupWindows)

Parameter:

IgnorePopupWindows: This parameter controls whether the command considers only application windows when deciding which window was last active. Valid settings are as follows:

Value	Description
0	Restores and activates the last window to have the focus, regardless of what type of window it is. This option is not recommended.
1	Restores and activates the last application window to have the focus that meets one of the following conditions: • The application window is modal and has a double border that may have been created with or without a title bar. • The application window is not a popup window unless it is a modal dialog, as defined by its window style.

Example:

```
' Go Dial.
CallMakeAppActiveLast(1)
```

Notes:

- You must not put the **CallMakeAppActiveLast** command in a user action and then assign the same action as the Go Dial Action in the Call Control tab of the Options dialog, otherwise the user action will continuously attempt to call itself.
- You can find out whether it is possible to make a new call at your extension by interrogating the value of the [**CanCallDial**] macro variable.

14.29 CallMakeInput

CallMakeInput

This command shows an input box in which the user enters a telephone number to be . A default value can be specified which is initially shown in the input box.

If the extension device is in a state that cannot facilitate making a call, an error occurs.

Syntax:

CallMakeInput(Extension, DefaultDialStr, AutoPrefix)

Parameters:

- **Extension**: The extension device to make a new call at. If a blank string is specified, then the extension assigned to the running instance of will be used.
- **DialString**: The telephone number to dial to display by default in the input box.
- **AutoPrefix**: This is a numerical argument that when set to “0” dials the digits exactly as entered in the input box by the user.

  When this value is set to “1”, the number to be is affected by the dial rules configured within . At a minimum this means that the outbound dial prefix will be included in the digits sent to the telephone system.

Example:

```
' Prompt the user to make a new call.
' The initial default is set blank.
CallMakeInput(“”, “”, 1)
```

Notes:

- See the **CallMake** command for information on special characters that can be entered in the number to be , depending on telephone system.
- You can find out whether it is possible to make a new call at your extension by interrogating the value of the [**CanCallDial**] macro variable.
- You can generically find the device number of the extension associated with the current installation of by using the [**LocalExtension**] macro variable.

14.30 CallMonitor

CallMonitor

This command monitors, at a specified device, an external trunk line call active on the given extension.

If an external answered trunk line call does not exist at the extension device depicted by the value in the “exttarget” argument, the **CallMonitor** macro statement will generate an error.

Syntax:

CallMonitor(Extension, ExtTarget, MonitorType)

Parameters:

- **Extension**: The extension device of the supervisor who will monitor the target call. If this is a blank string, the target call will be monitored at the extension currently assigned to .
- **ExtTarget**: The extension where the external trunk line call is active. This call at this device will be monitored using the monitor type specified in the “MonitorType” parameter.
- **MonitorType**: This numerical value defines the type of monitoring to perform, as follows:

Value	Description
0	**Silent Monitor**: This allows an agent group supervisor to listen in on an agent’s conversation from the supervisor extension. No indication is made to the agent or extension that is being monitored unless specified in the telephone system’s programming.
1	
2	

Example:

```
' Silent monitor the active call at extension
' 210 from Supervisor extension 200.
CallMonitor(“200”, “210”, 0)
```

Notes:

- ertain or all types of call monitoring capability may be disabled on the telephone system due to local bylaws or regulations, etc.
- You can generically find the device number of the extension associated with the current installation of by using the [**LocalExtension**] macro variable.

14.31 CallPage

CallPage

This command performs a page to the specified page group from the given extension.

If the extension device is in a state that cannot facilitate making a call, an error occurs. Alternatively, another device may be calling the same page group in which case an error may also occur depending on the telephone system in use.

Syntax:

CallPage(Extension, PageGroup)

Parameters:

- **Extension**: The extension device to perform the page from. If this is a blank string, the page will be performed at the extension currently assigned to .
- **PageGroup**: The group to be paged.

Example:

```
' Page from current extension
' to page group 10.
CallPage("","10")
```

Notes:

- You can find out whether it is possible to make a call from your extension by interrogating the value of the [**CanCallDial**] macro variable.
- You can generically find the device number of the extension associated with the current installation of by using the [**LocalExtension**] macro variable.

14.32 CallPickup

CallPickup

This command picks up a call that is camped-on (queued) or alerting at a target device on the given extension.

If the extension device is in a state that cannot facilitate making a call, then an error occurs. If there are no calls alerting or camped-on to the device specified then no errors will be generated and the pickup request is ignored.

Syntax:

CallPickup(Extension, AlertingDevice)

Parameters:

- **Extension**: The extension device to perform to pickup the call at. If this is a blank string, the call will be picked up at the extension currently assigned to .
- **AlertingDevice**: The extension that has a queued or alerting call that is to be picked up at the “Extension” device.

Example:

```
' Pick up the first call that is
' alerting or camped-on extension 1000.
CallPickup(“”,“1000”)
```

Notes:

- You can find out whether it is possible to make a call from your extension by interrogating the value of the [**CanCallDial**] macro variable.
- You can generically find the device number of the extension associated with the current installation of by using the [**LocalExtension**] macro variable.

14.33 CallRecord

CallRecord

This command records an external trunk line call at the given extension. The call is recorded into the voice mailbox specified in the command's parameters.

If an external answered trunk line call does not exist at the extension device specified, the command will generate an error.

Syntax:

CallRecord(ExtTarget, CallItem, Mailbox)

Parameters:

- **ExtTarget**: The extension device that has the active call to be recorded. If this is a blank string, the call will be recorded at the extension currently assigned to .
- **CallItem**: The index of the call to record. Calls in the list are identified as 1 for the first call in the call list, 2 for the second call etc. A value of 0 will instruct to automatically record the first external trunk line call at the given extension.

  In fact, if the call specified at the given extension is not an external trunk line call, the next call at the extension that is an external trunk line call is used instead.
- **Mailbox**: The voice mailbox that the call should be recorded to.

Example:

```
' Record the call at extension 200 to
' Voice Mailbox 201.
CallRecord("200", 0, "201")
```

Notes:

-
- The "Mailbox" argument is completely ignored when the target extension's Voice Mail Information setting "User-Keyed Mailbox" is disabled in the telephone system's programming.
- You can generically find the device number of the extension associated with the current installation of by using the [LocalExtension] macro variable.

14.34 CallRetrieve

CallRetrieve

This command retrieves an exclusively held call at the given extension.

If there is no exclusively held call at the specified extension or the extension device is in a state that cannot facilitate retrieving a held call, then an error occurs.

Syntax:

CallRetrieve(Extension, CallItem)

Parameters:

- **Extension**: The extension device that has the held call to be retrieved. If this is a blank string, then the call will be recorded at the extension currently assigned to .

- **CallItem**: The index of the call to retrieve. Calls in the list are identified as 1 for the first call in the call list, 2 for the second call etc. A value of 0 will instruct to automatically retrieve the first exclusively held call at the given extension.

  In fact, if the call specified at the given extension is not an exclusively held call, then an error is generated.

Example:

```
' Retrieve a call from exclusive
' hold at the current extension.
CallRetrieve("", 0)
```

Notes:

- You can find out whether it is possible to retrieve an exclusively held call at your extension by interrogating the value of the [**CanCallRetrieve**] macro variable.

- You can generically find the device number of the extension associated with the current installation of by using the [**LocalExtension**] macro variable.

14.35 CallSelect

CallSelect

This command highlights the given call in the active call list window of . The selected call is used to decide which call's information is returned when referring to call related macro variables

Syntax:

CallSelect(CallItem)

Parameter:

CallItem: The index of the call to select, where 1 represents the first call in the list, 2 represents the second call, and so on. Specifying a call item of 0 will remove any selection from the call list.

Example:

```
' Select the first call in the call list
CallSelect(1)
```

Notes:

- If you use this command in a user action that is fired because of a rule, the call specific macro variables will reflect the information from the new selected call once this command has been called. However, the [**CallSource**] macro variable will remain equal to the original call that caused the rule to fire. The [**CallSelected**] macro variable will take the value of the new selected call in the call list.

- You usually need to use the **CallSelect** command when you have written an action that makes a new call and you need to refer to information relating to the new call in the same action (for example, by using macro variables or macro commands such as the **ExitMacroIfCallType** or **GotoIfCallType** macro commands).

- Some of the "Look and Feels" in do not have an active call list window. In such a scenario, the selected call is still changed, but there will be no change in the user interface.

14.36 CallTransfer

CallTransfer

This command places the given call that is currently in the answered state on exclusive hold, before making a new announcement call to the given telephone number.

If the extension device is in a state that cannot facilitate initiating a call transfer, an error occurs.

Syntax:

CallTransfer(Extension, DialString, AutoPrefix)

Parameters:

- **Extension**: The extension to perform the transfer at. If the extension is a blank string, then the device currently associated with is used instead.
- **DialString**: The telephone number to make the announcement call to. If this is a blank string then the user will be prompted for the number to transfer to.
- **AutoPrefix**: This is a numerical argument that when set to “0” dials the digits exactly as entered in the “DialString” parameter.

  When this value is set to “1”, the number to be is affected by the dial rules configured within . At a minimum this means that the outbound dial prefix will be included in the digits sent to the telephone system (if the telephone number appears to be external).

Example:

```
' Get the number to dial from the user.
InputBox(“Enter number :”,“Transfer”,“”, 1)
' End macro if nothing was entered.
ExitMacroStrValue([Data1], “”, 1)
' Exclusively hold the current call and
' make a new call using the number entered.
CallTransfer(“”,[Data1], 1)
```

Notes:

- You can find out whether it is possible to initiate a call transfer at your extension by interrogating the value of the [**CanCallTrans**] macro variable.
- You can generically find the device number of the extension associated with the current installation of by using the [**LocalExtension**] macro variable.

14.37 CallTransferComplete

CallTransferComplete

This command completes a call transfer at the given extension. For the command to be able to work there must already be the call-to-transfer on exclusive hold at the specified extension. There must also be a previously set up consultation call that is in the answered state. The command transfers the specified held call to the party at the distant end of the currently answered consultation call.

If the extension device is in a state that cannot facilitate completing a call transfer, an error occurs.

Syntax:

CallTransferComplete(Extension, CallItem)

Parameters:

- **Extension**: The extension device that has the consultation call to be completed. If this is a blank string, then the transfer will be completed at the extension currently assigned to.

- **CallItem**: The index of the held call to transfer to the answered consultation call. Calls in the list are identified as 1 for the first call in the call list, 2 for the second call etc. A value of 0 will instruct to automatically transfer the first held call at the given extension.

Example:

```
' Complete a call transfer at the current
' extension.
CallTransferComplete("", 0)
```

Notes:

- You can find out whether it is possible to complete a call transfer at your extension by interrogating the value of the [**CanCallTransComp**] macro variable.

- You can generically find the device number of the extension associated with the current installation of by using the [**LocalExtension**] macro variable.

14.38 CallTransRedir

CallTransRedircmd

This command performs a blind (direct) transfer of an answered or alerting call at the given device to another party. The command prompts the user for the party to transfer the call to.

If there is no moveable call at the specified extension, or the extension device is in a state that cannot facilitate transferring or redirecting a call, an error occurs.

Syntax:

CallTransRedir(Extension, CallItem)

Parameters:

- **Extension**: The extension device that has the call to be blind transferred. If this is a blank string, the transfer will be performed at the extension currently assigned to .
- **CallItem**: The index of the call to blind transfer. Calls in the list are identified as 1 for the first call in the call list, 2 for the second call etc. A value of 0 will instruct to automatically transfer the first alerting, answered, or external outbound call at the given extension.

Example:

```
' Directly transfer or redirect the first call
' item (answered or alerting).
' The user will automatically be prompted for
' the party to transfer/redirect the call to.
CallTransRedir("", 1)
```

Notes:

- When the user enters in the number of another party to move the call to, they do not need to specify the outbound dial prefix or long distance dial code at the beginning of the target party's dial string. These are automatically added to the beginning of the dial string using the rules contained within .
- You can find out whether it is possible to move a call at your extension by interrogating the value of the [**CanCallTransRedir**] macro variable.
- You can generically find the device number of the extension associated with the current installation of by using the [**LocalExtension**] macro variable.

14.39 CallTransRedirDirect

CallTransRedirDirect

This command performs a blind (direct) transfer of an answered or alerting call at the given device to another party specified in the command's parameters.

If the given call is alerting, it will be redirected to the specified party. If the call is answered, it will be blind transferred. The call always alerts the party that it is transferred to immediately after the command is completed.

If there is no moveable call at the specified extension, or the extension device is in a state that cannot facilitate transferring or redirecting a call, then an error occurs.

Syntax:

CallTransRedirDirect(Extension, CallItem, DialString)

Parameters:

- **Extension**: The extension device that has the call to be blind transferred. If this is a blank string, then the transfer will be performed at the extension currently assigned to .
- **CallItem**: The index of the call to blind transfer. Calls in the list are identified as 1 for the first call in the call list, 2 for the second call etc. A value of 0 will instruct to automatically transfer the first alerting, answered, or external outbound call at the given extension.
- **DialString**: The number of the party to transfer the call to.

Example:

```
' Redirects the currently selected call to a specific device.
CallTransRedirDirect("", [CallSelected], "20000")
```

Notes:

- The number of the party specified in the "DialString" argument does not need to include the outbound dial prefix or long distance dial code. These are automatically added to the beginning of the number using the dial rules contained in .
- You can find out whether it is possible to move a call at your extension by interrogating the value of the [**CanCallTransRedir**] macro variable.
- You can generically find the device number of the extension associated with the current installation of by using the [**LocalExtension**] macro variable.

14.40 ClipboardAppendText

ClipboardAppendText

This command adds the given text to the end of the current Windows clipboard.

For example, if the clipboard contained “”, and this command was called with “ Developer SDK” as the parameter, the clipboard would contain “ Developer SDK” after the command had completed.

Syntax:

ClipboardAppendText(Text)

Parameter:

Text: The text to append to the current clipboard contents.

Example:

```
' Append line number of current
' call to the clipboard text.
ClipboardAppendText(“Line : ” + [Line])
```

Notes:

- If the clipboard contains a non-text based format prior to the command being called, the clipboard will be cleared, and the given text stored in it instead.
- You can obtain the text currently in the clipboard by using the [**Clipboard**] macro variable.

14.41 ClipboardSetText

ClipboardSetText

Sets the Windows clipboard contents to be the given text.

Syntax:

ClipboardSetText(Text)

Parameter:

Text: The text to set the current clipboard contents to.

Example:

```
' Copy line number of current call to clipboard.
ClipboardSetText("Line : " + [Line])
```

Notes:

- This command will clear the contents of the clipboard before storing the given text in it.
- You can obtain the text currently in the clipboard by using the [**Clipboard**] macro variable.

14.42 DataSetNum

DataSetNum

Stores a numerical value in a particular [**Data*n***] macro variable.

Syntax:

DataSetNum(DataVar, NumValue)

Parameters:

- **DataVar**: A numerical value between 1 and 11 which depicts the [**Data*n***] variable to store the “NumValue” value in, where 1 represents [Data1], 2 depicts [Data2], etc.

- **NumValue**: The numerical value to store in the macro variable. This can also be an expression that when evaluated would return a numerical value, e.g., “(10 * 3) + 5”

Example:

```
' Set the [Data1] Macro Variable Reference
' to a value of 30.
DataSetNum(1, 30)
```

Notes:

- The [**Data*n***] macro variables only hold their value for the life of the execution of the user action. After the action has completed this execution, the macro variable settings are forgotten.

- If you try to use a textual value in the “NumValue” argument, your user action will not compile. You should use DataSetStr if you want to store a textual value in a data macro variable.

14.43 DataSetStr

DataSetStr

Stores a textual (string) value in a particular [**Data*n***] macro variable.

Syntax:

DataSetStr(DataVar, Text)

Parameters:

- **DataVar**: A numerical value between 1 and 11 which depicts the [**Data*n***] variable to store the “Text” value in, where 1 represents [Data1], 2 depicts [Data2], etc.

- **Text**: The textual (string) value to store in the macro variable. This can also be an expression that, when evaluated, would return a text-based value, for example,

  “” + “ ” + “Developer SDK”

Example:

```
' Temporarily sets the [Data1] Macro Variable
' to the current text value in the clipboard.
DataSetStr(1, [Clipboard])
```

Notes:

- The [**Data*n***] macro variables only hold their value for the life of the execution of the user action. After the action has completed this execution, the macro variable settings are forgotten.

- If you try to use a numeric value in the “Text” argument, your user action will not compile. You should use **DataSetNum** if you want to store a numerical value in a data macro variable.

14.44 DataSetStrChrReplace

DataSetStrChrReplace

This command replaces all sub-strings in a given string with a particular replacement string, and then stores the result in a [**Data*n***] macro variable.

Syntax:

DataSetStrChrReplace(DataVar, Text, TextToFind, TextReplacement)

Parameters:

- **DataVar**: A numerical value between 1 and 11 which depicts the [**Data*n***] variable to store the result of the string replacement, where 1 represents [**Data1**], 2 depicts [**Data2**], etc.
- **Text**: The original textual value to perform the string replacement in.
- **TextToFind**: The sub-string that will be replaced in the “Text” value. Any occurrences of this sub-string will be replaced, not just the first one.
- **TextReplacement**: The string to replace occurrences of “TextToFind” with. This string does not need to be the same length as “TextToFind”.

Note: The TextReplacement function is case-sensitive.

Example:

```
' The following macro works around the problem where you need to
' send a string as keystrokes that contains the “(” or “)”
' characters, which have a special meaning to Sendkeys.
' Copy the contents of Column 3 for the current call to
' [Data1].
DataSetStr(1, [Col3])
' Now replace any “(” or “)” characters with
' their appropriate Sendkeys syntax (bracket
' surrounded by braces)
DataSetStrChrReplace(1, [Data1], “(“, “{(}”)
DataSetStrChrReplace(1, [Data1], “)”, “{)}”)
' Send the keystrokes.
Sendkeys([Data1])
```

Notes:

The [**Data*n***] macro variables hold their value only for the life of the execution of the user action. After the action has completed this execution, the macro variable settings are forgotten.

14.45 DataSetStrChrStrip

DataSetStrChrStrip

This command removes all occurrences of characters in a sub-string from a given string, and stores the result in a [**Data*n***] macro variable.

For example, if the string “No vowels in this” was provided along with a string of characters to remove as “aeiou”, the resulting string would be “N vwls n ths”.

Syntax:

DataSetStrChrStrip(DataVar, StrValue, StripChars)

Parameters:

- **DataVar**: A numerical value between 1 and 11 which depicts the [**Data*n***] variable to store the result of the string after characters are stripped, where 1 represents [**Data1**], 2 depicts [**Data2**], etc.
- **StrValue**: A textual value that will have characters contained in “StripChars” removed from it, before the result in stored in a macro variable.
- **StripChars**: A string of characters that must be removed from “StrValue”. Any character that occurs in this string will be removed from “StrValue;” if the character occurs multiple times in “StrValue,” it will be removed multiple times.

Example:

```
' Copy the text selected in the last active
' application window, and strip any non-numeric
' characters from the string.
'
' Set focus to last active application.
AppActivateLastFoc(1)
' Ensure focus has changed.
YieldToOs
' Copy the selected text.
AppCopyText
' Now strip a-z (lower case) characters
DataSetStrChrStrip(1, [Clipboard], "abcdefghijklmnopqrstuvwxyz")
' Now strip A-Z (upper case) characters
DataSetStrChrStrip(1, [Data1], "ABCDEFGHIJKLMNOPQRSTUVWXYZ")
```

Notes:

The [**Data*n***] macro variables only hold their value for the life of the execution of the user action. After the action has completed this execution, the macro variable settings are forgotten.

14.46 DataSetStrLeft

DataSetStrLeft

This command stores the leftmost part of a given string in a [**Data*n***] macro variable, specifying how many characters of the string depict the leftmost part.

Syntax:

DataSetStrLeft(DataVar, StrValue, LeftPart)

Parameters:

- **DataVar**: A numerical value between 1 and 11 which depicts the [**Data*n***] variable to store the result of the string after processing, where 1 represents [Data1], 2 depicts [Data2], etc.
- **StrValue**: A textual value that will have the first “LeftPart” characters copied into the given [**Data*n***] macro variable.
- **LeftPart**: The number of characters to copy from the start of the “StrValue” string into the [**Data*n***] macro variable.

Example:

```
' Assign the first 5 digits (area prefix) of a
' telephone number to the [Data1] Macro
' Variable Reference.
DataSetStrLeft(1, [Digits], 5)
```

Notes:

The command does not affect the contents of “StrValue,” since the characters are copied into the [**Data*n***] macro variable, rather than moved.

The [**Data*n***] macro variables only hold their value for the life of the execution of the user action. After the action has completed this execution, the macro variable settings are forgotten.

14.47 DataSetStrLen

DataSetStrLen

This command stores the length of the given string in a specific [**Data*n***] macro variable.

Syntax:

DataSetStrLen(DataVar, StrValue)

Parameters:

- **DataVar**: A numerical value between 1 and 11 which depicts the [**Data*n***] variable to store the length of the string in, where 1 represents [Data1], 2 depicts [Data2], etc.
- **StrValue**: A textual value to calculate the length of, and store in the given [**Data*n***] macro variable.

Example:

```
' Assign the length of the telephone number
' to the [Data1] Macro Variable Reference.
DataSetStrLen(1, [Digits])
```

Notes:

The [**Data*n***] macro variables only hold their value for the life of the execution of the user action. After the action has completed this execution, the macro variable settings are forgotten.

14.48 DataSetStrMid

DataSetStrMid

This command copies a range of characters from a given string into a specific [**Data*n***] macro variable.

Syntax:

DataSetStrMid(DataVar, StrValue, Start, Length)

Parameters:

- **DataVar**: A numerical value between 1 and 11 which depicts the [**Data*n***] variable to store the copied characters in, where 1 represents [Data1], 2 depicts [Data2], etc.
- **StrValue**: A textual value from which a range of characters will be copied and stored in the given [**Data*n***] macro variable.
- **Start**: he index of the first character in “StrValue” to begin copying from, where a value of 1 represents the first character, 2 the second, and so on.
- **Length**: The number of characters in “StrValue” to copy.

Example:

```
' Obtain the 7th up to the 10th character from
' the clipboard.
DataSetStrMid(1, [Clipboard], 7, 4)
```

Notes:

- If either “Start” or “Length” are out of a valid range, e.g., greater than the length of the string, an error does not occur, and the [**Data*n***] macro variable is set correctly. For example, passing a “Length” of 40 when the original string had only 20 characters in it, would return all characters from the specified “Start” character.
- The [**Data*n***] macro variables only hold their value for the life of the execution of the user action. After the action has completed this execution, the macro variable settings are forgotten.

14.49 DataSetStrRight

DataSetStrRight

This command stores the rightmost part of a given string in a [**Data*n***] macro variable, specifying how many characters of the string depict the rightmost part.

Syntax:

DataSetStrRight(DataVar, StrValue, RightPart)

Parameters:

- **DataVar**: A numerical value between 1 and 11 which depicts the [**Data*n***] variable to store the result of the string after processing, where 1 represents [Data1], 2 depicts [Data2], etc.
- **StrValue**: A textual value that will have the right “RightPart” characters copied into the given [**Data*n***] macro variable.
- **RightPart**: The number of characters to copy from the end of the “StrValue” string into the [**Data*n***] macro variable.

Example:

```
' Take the last 3 characters from the
' digits of the current call and assign them
' to the [Data1] Macro Variable Reference.
DataSetStrRight(1, [DDIDigits], 3)
```

Notes:

The [**Data*n***] macro variables only hold their value for the life of the execution of the user action. After the action has completed this execution, the macro variable settings are forgotten.

14.50 DDEClose

DDEClose

This command terminates an active DDE conversation on the specified channel. If the given channel does not have an active DDE conversation, an error occurs. If you have used the SetErrorsFatal command to switch off error handling, then the error will be accessible from the [ErrorDesc] and [ErrorDesc] macro variables, otherwise macro execution will stop.

Syntax:

DDEClose(Channel)

Parameter:

Channel: The channel number of the DDE conversation that should be terminated. This can be a value from 1 to 6.

Example:

```
' Close DDE conversation on channel 1.
DDEClose(1)
```

Notes:

When execution reaches the end of a user action, all open DDE conversations are closed anyway.

14.51 DDEOpen

DDEOpen

Attempts to start a DDE conversation on the channel specified. If a conversation is already active on the channel, an error occurs .If you have used the SetErrorsFatal command to switch off error handling, then the error will be accessible from the [ErrorDesc] and [ErrorNum] macro variables, otherwise macro execution will stop.

The DDE topic and DDE application name for the conversation must have already been set using commands **DDESetAppName** and **DDESetTopic**.

Syntax:

DDEOpen(Channel)

Parameters:

Channel: The channel number to open the DDE conversation on. This can be a value from 1 to 6.

Example:

```
DDEOpen(1)
```

Notes:

To start a DDE conversation with another application, that application should be running already. To automatically start an application before you try to start communicating with it, use the **AppActivateLikeShell** or **Shell** commands.

14.52 DDEPoke

DDEPoke

This command sends a piece of textual data related to a named item over an active DDE conversation. This is typically used to set a variable or object with a particular value when communicating via DDE, e.g., to set the contents of a cell in Excel.

If an error occurs and you have used the SetErrorsFatal command to switch off error handling, then the error will be accessible from the [ErrorDesc] and [ErrorNum] macro variables, otherwise macro execution will stop.

Syntax:

DDEPoke(Channel, ItemName, DataString)

Parameters:

- **Channel**: The channel number of the DDE conversation to send data over. This can be a value from 1 to 6.
- **ItemName**: The name of the item to set in the application with which a DDE conversation is active. The value of this parameter will depend entirely on the application being conversed with, and the topic on which the conversation is about.
- **DataString**: The data to send across to the application with which a DDE conversation is active.

Example:

```
' Place value 100 in cell (row 1, column 1)
' in Excel spreadsheet.
DDEPoke(1, "R1C1", "100")
```

14.53 DDERequest

DDERequest

This command requests a piece of data related to a named item from an active DDE conversation. This is typically used to retrieve information about a current variable or object when communicating via DDE, e.g., to get the contents of a cell in Excel.

If the request is successful, the result is returned in the appropriate [**DDE*n***] macro variable for the DDE channel being used, e.g., if the conversation is on channel 2, the result is returned in the [**DDE2**] macro variable.

If an error occurs and you have used the SetErrorsFatal command to switch off error handling, then the error will be accessible from the [ErrorDesc] and [ErrorNum] macro variables, otherwise macro execution will stop.

Syntax:

DDERequest(Channel, ItemName)

Parameters:

- **Channel**: The channel number of the DDE conversation to request data from. This can be a value from 1 to 6.
- **ItemName**: The name of the item to retrieve in the application with which a DDE conversation is active. The value of this parameter will depend entirely on the application being conversed with, and the topic on which the conversation is about.

Example:

```
' Request all the DDE topics that Microsoft
' Word supports.
DDESetAppName(1, "WinWord")
DDESetTopic(1, "System")
DDEOpen(1)
DDERequest(1, "Topics")
' Show all the topics to the user.
MessageBox([DDE1], 0, "")
DDEClose(1)
```

Notes:

- If a conversation is not active on the given channel when a DDE request is made, an error will be returned.
- If you start a conversation with an application's "System" topic, you can usually retrieve a list of all topics currently supported by that application.

14.54 DDESendCmd

DDESendCmd

This command sends an application-specific command string over a given DDE conversation. This command can be used when a DDE server supports receiving commands to make it perform actions; the commands that can be passed to the DDE server are dependent upon the server. Consult the other application's documentation for further information.

If an error occurs and you have used the SetErrorsFatal command to switch off error handling, then the error will be accessible from the [ErrorDesc] and [ErrorNum] macro variables, otherwise macro execution will stop.

Syntax:

DDESendCmd(Channel, CommandString)

Parameters:

- **Channel**: The channel number of the DDE conversation to send a command to. This can be a value from 1 to 6.
- **CommandString**: The application-specific command to send to the DDE server.

Example:

```
' Open Excel spreadsheet ORDERS.XLS.
DDESendCmd(1, "[OPEN(""ORDERS.XLS"")]")
```

Notes:

This command will return an error if the DDE server returns an error because the command failed, or the command was invalid in some form. In such instances, the use of the **GotoIfDDESendCmd** command is preferred, as it provides conditional branching based on whether the DDE command was successful or not.

14.55 DDESetAppName

DDESetAppName

This command sets the DDE application name that the given channel will use when initiating a conversation with the **DDEOpen** command.

The application name must be set for a channel before you set the topic name for that channel with the **DDESetTopic** command.

Syntax:

DDESetAppName(Channel, AppName)

Parameters:

- **Channel**: The channel number of the DDE conversation to set the application name for. This can be a value from 1 to 6.
- **AppName**: The application name that you want to hold a conversation with. The name is dependent on the application that you want to communicate with, e.g., 's application name is "Callview".

Example:

```
' Set the application name on DDE channel 1
' to communicate with Microsoft Excel.
DDESetAppName(1, "EXCEL")
```

Notes:

You cannot change the application name for a conversation after the conversation has started. You must call **DDEClose** before the application name can be changed.

14.56 DDESetTimeOut

DDESetTimeOut

This command sets the timeout period for DDE commands on a given channel. A timeout can occur on a DDE channel if the application being conversed with takes a long time to respond to a DDE command, e.g., a search for a record could take a long time, and so the DDE client has to assume after a given time that the DDE server is not going to respond.

Syntax:

DDETimeOut(Channel, Timeout)

Parameters:

- **Channel**: The channel number of the DDE conversation to set the timeout for. This can be a value from 1 to 6.
- **Timeout**: A numerical value which depicts the DDE timeout in tenths of a second. So setting this value to 10 would result in a DDE timeout of 10 x 0.1s = 1s. The default value is 100, providing a 10 second DDE timeout.

Example:

```
' Set DDE timeout to half a second.
DDETimeOut(1, 5)
```

Notes:

- Under normal circumstances there is no need to change the DDE timeout. If you are sending a command to a DDE server that you expect may take some time, then it is sensible to increase the DDE timeout before sending the command.
- You can change the DDE timeout even when a conversation is active, although it is a good idea to set it before opening the conversation as well.
- If a DDE command times out, then an error will occur and your user action will stop. You can control whether timeouts return an error using the **DDESetTimeOutWarningOff** command.
- The DDE timeout is returned to its default value when the user action ends.

14.57 DDESetTimeOutWarningOff

DDESetTimeOutWarningOff

This command specifies whether an error is generated when a DDE command times out. By default DDE timeouts will generate an error.

Syntax:

DDESetTimeOutWarningOff(Channel, DisableTimeout)

Parameters:

- **Channel**: The channel number of the DDE conversation to set the timeout warning setting for. This can be a value from 1 to 6.

- **DisableTimeout**: If this numerical value is 1, timeout warnings are disabled, and so no errors will be generated.

  If the value is 0, timeout warnings are enabled, and so errors will be generated if a timeout occurs.

Example:

```
' Set time-out warnings off for DDE channel 1.
DDESetTimeOutWarningOff(1, 1)
```

Notes:

You can enable or disable timeout warnings regardless of whether a DDE conversation is active or not. For example, if you knew that a particular DDE command was going to take an exceptionally long time, you would switch off the timeout warning before executing the DDE command, and then enable the timeout warning after the command had completed.

14.58 DDESetTopic

DDESetTopic

This command sets the DDE topic name that the given channel will use when initiating a conversation with the **DDEOpen** command.

The topic name must be set for a channel before you open a DDE conversation with **DDEOpen**. You must set the application name for the channel with DDESetAppName before you call this method.

Syntax:

DDESetTopic(Channel, TopicName)

Parameters:

- **Channel**: The channel number of the DDE conversation to set the topic name for. This can be a value from 1 to 6.
- **TopicName**: The topic name that you want to hold a conversation with. The name is dependent on the application that you want to communicate with. An application can support multiple topic names, but all DDE servers support the “System” topic.

Example:

```
' Set the topic name on DDE channel 1 to communicate
' with Microsoft Excel spreadsheet ORDERS.XLS.
DDESetTopic(1, “ORDERS.XLS”)
```

Notes:

You cannot change the topic name on a channel once a DDE conversation has been started on that channel. Close the conversation first with **DDEClose**.

14.59 End

End

This command stops execution of the user action. It is equivalent to execution reaching the end of the action, although you do not need to call **End** at the end of the action script.

Syntax:

End

Parameters:

None.

Example:

```
' Goto line label “AccessOpen” if Microsoft
' Access is currently running.
GotoIfAppActive(“AccessOpen”, “Microsoft Access”, 0)
End ' Exit macro if Access is not running.
AccessOpen:
' Continue with execution here...
```

14.60 ExitMacroAppActive

ExitMacroAppActive

This command stops execution of the user action based on whether an application window with a specific titlebar text is found or not.

Syntax:

ExitMacroAppActive(WindowTitle, IsRunning)

Parameters:

- **WindowTitle**: The left part of the title of the application window to find.

  The name that appears in the titlebar need not be fully specified. For instance, “Calc” would still find an open application with titlebar text “Calculator”. The comparison is also not case sensitive; i.e., “Calculator” and “calculator” appear identical.

- **IsRunning**: If this numerical value is 0, then the command will stop execution if the no window can be found with the given window title.

  If the value is 1, then the command will stop execution if a window is found with the given window title.

Example:

```
' Exit macro if Notepad application is open.
ExitMacroAppActive(“Notepad”, 1)
```

Notes:

- This command is useful for stopping actions that require an application to be open, e.g., an action that uses DDE to communicate with another application; if the application is not running then the DDE communication will not work.

- An alternative to this command is the **GotoIfAppActive** command, that allows you to conditionally branch depending on whether the window is open or not. This allows more flexibility than just ending the action.

14.61 ExitMacroIfCallType

ExitMacroIfCallType

This command stops execution of a user action if the current call is of a particular type, e.g., internal, or answered, etc.

Syntax:

ExitMacroIfCallType(CallType)

Parameter:

CallType: A numerical value that defines the type of call to check for. If the current call matches the specified type, the user action will stop its execution. The following options are available:

Value	Call Type
0	Internal call
1	External (trunk line) call
2	Outbound call
3	Inbound call
4	Held call
5	Unheld call
6	Answered call
7	Unanswered call
8	The telephone number or e-mail address was identified against ‘s telephone data import.
9	The telephone number or e-mail address was not identified against ‘s telephone data import. **Note**: If no is received, the number is considered not identified.
10	I was received for an inbound external call
11	was not received for an inbound external call

Example:

```
' Exit macro if the current call was received
' with no calling line identity ( ).
ExitMacroIfCallType(11)
```

Notes:

An alternative to this command is the **GotoIfCallType** command that allows you to conditionally branch depending on the type of call. This allows more flexibility than just ending the action.

14.62 ExitMacroIfNoCalls

ExitMacroIfNoCalls

This command stops execution of the user action based on the number of calls active at the extension associated with .

Syntax:

ExitMacroIfNoCalls(CallCount)

Parameter:

CallCount: A numerical value that depicts the number of active calls that must be active at the extension assigned to for the user action to stop execution. For example, if this value is 0, the action will stop if there are no calls active at the assigned extension.

Example:

```
' Exit macro if there are no calls.
ExitMacroIfNoCalls(0)
```

Notes:

- This command is useful for stopping user actions that contain call-based macro variables. Such macro variables return an error if no call is active, so stopping the action before the macro variables are referenced can be very helpful.
- An alternative to this command is the **GotoIfNoCalls** command that allows you to conditionally branch depending on the number of calls. This allows more flexibility than just ending the action.

14.63 ExitMacroNumValue

ExitMacroNumValue

This command stops execution of a user action depending on the comparison between two numerical expressions.

Syntax:

ExitMacroNumValue(Value1, Value2, CompareType)

Parameters:

- **Value1**: The first numerical value that will be compared against the second numerical value, “Value2.”
- **Value2**: The second numerical value that will be compared against the first value, “Value1.”
- **CompareType**: If this value is 0, the command will stop action execution if “Value1” and “Value2” are different.

  If the value is 1, the command stops execution if the two values are equal.

Example:

```
' Stop macro if current call is on line 700.
ExitMacroNumValue([Line], 700, 1)
```

14.64 ExitMacroStrValue

ExitMacroStrValue

This command stops execution of a user action depending on the comparisons of two string expressions.

Syntax:

ExitMacroStrValue(String1, String2, CompareType)

Parameters:

- **String1**: The first string value that will be compared against the second string value, “String2.”
- **String2**: The second string value that will be compared against the first string, “String1.”
- **CompareType**: If this value is 0, then the command will stop action execution if “String1” and “String2” are different.

  If the value is 1, the command stops execution if the two strings are equal.

Example:

```
' Stop macro if the current call was received on
' the sales order telephone number.
ExitMacroStrValue([DNIS], “Sales Order Line”, 1)
```

Notes:

When comparing strings, this command ignores case, i.e., “Hello” is the same as “hello” which is the same as “HELLO”.

14.65 FileClose

FileClose

This command closes a file that has been opened using the FileOpen command.

Syntax:

FileClose(FileIndex)

Parameters:

FileIndex: This numeric value depicts which file handle to close. It can be a value between 1 and 5. When you open a file, you choose which of the 5 handles to use, and then specify that handle in all further file operations until the file is closed.

Example:

```
' Close file handle 1
FileClose(1)
```

Notes:

- All open file handles are closed when the macro exits.
- By default the file handling commands will halt execution of the macro if an error occurs. You can use the SetErrorsFatal command to stop this from occurring, so that the error description and number are available via the [ErrorDesc] and [ErrorNum] macro variables.

14.66 FileOpen

FileOpen

This command opens a file.

Syntax:

FileOpen(FileIndex, FileName, OpenFor)

Parameters:

- **FileIndex:** This numeric value depicts which file handle to open a file on. It can be a value between 1 and 5. When you open a file, you choose which of the 5 handles to use, and then specify that handle in all further file operations until the file is closed. If the handle you select is already being used with an open file, then an error is generated.

- **FileName:** This is the fully pathed filename of the file to open.

- **OpenFor:** This numeric value defines how the file will be opened. It can be a value between 1 and 3, as follows:

    1 Open the file for reading
    2 Open the file for writing, always creating a brand new file
    3 Open the file for appending, only creating a new file if the given file doesn't exist

Example:

```
' Open a file called "CallLog.Txt" in the Windows folder for
' appending, so that it will be created if it doesn't exist
FileOpen(1, [WINDIR] + "CallLog.Txt", 3)
```

Notes:

- Although all open file handles are closed when the macro exits, you should call the FileClose command once you are finished using a file or file handle.

- By default the file handling commands will halt execution of the macro if an error occurs. You can use the SetErrorsFatal command to stop this from occurring, so that the error description and number are available via the [ErrorDesc] and [ErrorNum] macro variables.

14.67 FileRead

FileRead

This command reads data from a file that has been opened using the FileOpen command. It reads a fixed number of bytes from the file into the given [Datan] macro variable.

Syntax:

FileRead(FileIndex, NumBytes, DataVar)

Parameters:

- **FileIndex**: This numeric value depicts which file handle to read data from. It can be a value between 1 and 5. When you open a file, you choose which of the 5 handles to use, and then specify that handle in all further file operations until the file is closed.
  If you have not opened the given file for reading, then an error will occur.

- **NumBytes**: This numeric value depicts the number of bytes to read from the given file. The bytes will be read directly into memory, and the file pointer advanced.

- **DataVar**: This numeric value depicts which of the [Datan] macro variables will receive the data that is read. This can be a value between 1 and 11.

Example:

```
' Read the next 6 bytes of a file into [Data3]
FileRead(1, 6, 3)
```

Notes:

- This command is best used for files where the data is stored in a fixed length format, rather than a variable length format. For variable length formats, use the FieldReadLine command.

- The [EOFn] macro variables will be set to a non-zero value when you have reached the end of the file.

- By default the file handling commands will halt execution of the macro if an error occurs. You can use the SetErrorsFatal command to stop this from occurring, so that the error description and number are available via the [ErrorDesc] and [ErrorNum] macro variables.

14.68 FileReadLine

FileReadLine

This command reads data from a file that has been opened using the FileOpen command. It reads an entire line of text from the file into the given [Datan] macro variable.

Syntax:

FileReadLine(FileIndex, DataVar)

Parameters:

- **FileIndex**: This numeric value depicts which file handle to read data from. It can be a value between 1 and 5. When you open a file, you choose which of the 5 handles to use, and then specify that handle in all further file operations until the file is closed.
  If you have not opened the given file for reading, then an error will occur.

- **DataVar**: This numeric value depicts which of the [Datan] macro variables will receive the data that is read. This can be a value between 1 and 11.

Example:

```
' Read the next line from the file into [Data2]
FileReadLine(1, 2)
```

Notes:

- This command is best used for files where the data is stored in a variable length format, rather than a fixed length format. For fixed length formats, use the FieldRead command.

- The [EOFn] macro variables will be set to a non-zero value when you have reached the end of the file.

- By default the file handling commands will halt execution of the macro if an error occurs. You can use the SetErrorsFatal command to stop this from occurring, so that the error description and number are available via the [ErrorDesc] and [ErrorNum] macro variables.

14.69 FileWrite

FileWrite

This command writes data to a file that has been opened using the FileOpen command. It writes a given string exactly as it is in memory to the file, so can be used to write non-alphanumeric characters to a file.

Syntax:

FileWrite(FileIndex, Data)

Parameters:

- **FileIndex**: This numeric value depicts which file handle to read data from. It can be a value between 1 and 5. When you open a file, you choose which of the 5 handles to use, and then specify that handle in all further file operations until the file is closed.
  If you have not opened the given file for writing or appending, then an error will occur.

- **Data**: This string value is the data to be written to the file. This value can be built up using non-alphanumeric characters, using such terminology as {5} + {254} + {131}.

Example:

```
' Writes 4 bytes of data to the file open on channel 1.
FileWrite(1, "XA" + {7} + {23})
```

Notes:

- This command is best used for files where the data is stored in a fixed length format, rather than a variable length format. For variable length formats, use the FieldWriteLine command.

- All open file handles are closed when the macro exits.

- By default the file handling commands will halt execution of the macro if an error occurs. You can use the SetErrorsFatal command to stop this from occurring, so that the error description and number are available via the [ErrorDesc] and [ErrorNum] macro variables.

14.70 FileWriteLine

FileWriteLine

This command writes a line data to a file that has been opened using the FileOpen command. The data is written to the file, and then appended with a new line character.

Syntax:

FileWriteLine(FileIndex, Data)

Parameters:

- **FileIndex**: This numeric value depicts which file handle to write data to. It can be a value between 1 and 5. When you open a file, you choose which of the 5 handles to use, and then specify that handle in all further file operations until the file is closed.
  If you have not opened the given file for writing or appending, then an error will occur.

- **Data**: This string value is the data to be written to the file. The data will be appended with a new line character when it is written.

Example:

```
' Write the selected call's telephone number to the open file
FileWriteLine(1, "Call from " + [Digits])
```

Notes:

- This command is best used for files where the data is stored in a variable length format, rather than a fixed length format. For fixed length formats, use the FieldWrite command.

- All open file handles are closed when the macro exits.

- By default the file handling commands will halt execution of the macro if an error occurs. You can use the SetErrorsFatal command to stop this from occurring, so that the error description and number are available via the [ErrorDesc] and [ErrorNum] macro variables.

14.71 FormatTelephoneNumber

FormatTelephoneNumber

This command formats a provided telephone number into a given style, as defined in the "Dial Formats" section of Callview's INI file, CALLVIEW32.INI.

The result is store into a given [Datan] macro variable, and can be utilized from there using the usual commands and macro variables.

This command is useful when searching a database for a telephone number, where the telephone number is stored in a formatted style, as opposed to a string of digits, e.g. stored as (01293) 608-200 as opposed to 01293608200.

Syntax:

FormatTelephoneNumber(Digits, Format, DataVar)

Parameters:

- **Digits**: This string value represents the telephone number to be formatted.
- **Format**: This numerical value represents which of the "Dial Formats" to format the number with. This can be between 1 and the value of the [TelNoFormatCount] macro variable.
- **DataVar**: This numeric value depicts the [Datan] macro variable that the formatted result will be stored in. For example, setting this value to 1 would store the property's value in the [Data1] macro variable.

Example:

```
' Format the current call's phone number using format 11,
' and store the result in [Data1].
FormatTelephoneNumber([Digits], 11, 1)
```

14.72 GetIniSetting

GetIniSetting

This command reads a single setting from the CVMACRO.INI file that can be used to provide special configuration for user-defined macros.

The setting is read into the given [Datan] macro variable, and can be utilized from there using the usual commands and macro variables.

Syntax:

GetIniSetting(SectionName, KeyName, Default, DataVar)

Parameters:

- **SectionName**: This string value represents the name of the section in the INI file where the setting is stored. Sections are denoted in INI files by wrapping them in square brackets, e.g. "[SectionName]".
- **KeyName**: This string value represents the name of the value in the INI file to read. The key to be read must fall within the section denoted by the "SectionName" parameter.
- **Default**: This string value represents the default value to store in the given data variable if they required value could not be located in the given section name.
- **DataVar**: This numeric value depicts the [Datan] macro variable that the retrieved value will be stored in. For example, setting this value to 1 would store the property's value in the [Data1] macro variable.

Example:

```
' Get the "SearchAllPhoneFields" setting from [Options] section
' into [Data3]. The default value is "1".
GetIniSetting("Options", "SearchAllPhoneFields", "1", 3)
```

Note:

You use the `SetIniSettingStr` and `SetIniSettingNum` commands to store information in the CVMACRO.INI file. You can also edit the file by hand using a text editor, such as Notepad.

14.73 GlobalDataGet

GlobalDataGet

This command retrieves a value from ’s global property set, and stores it in a given [**Data*n***] macro variable.

has a global property set that lasts for the duration that is connected to the . The global property set consists of several named properties that can all be accessed by any user action. This allows for sharing of information between different actions, or for an action to persist information between one execution and another.

Syntax:

GlobalDataGet(PropertyName, DataVar)

Parameters:

- **PropertyName**: This string value represents the name of an individual property that you want to get the current value for. If the property name does not exist, an error is generated.
- **DataVar**: This numerical value depicts the [**Data*n***] macro variable that the property’s current value will be stored in. For example, setting this value to 1 would store the property’s value in the [Data1] macro variable.

Example:

```
’ Get the “LastCallDate” property into [Data3]
GlobalDataGet(“LastCallDate”, 3)
```

Notes:

- You can also use the LocalDataGet command to get data from the action’s local property set. The local property set only contains properties for the current action while it is executing, and can be useful for extending storage beyond the 11 [**Data*n***] macro variables.
- You use the GlobalDataSetStr and GlobalDataSetNum commands to store information in the global property set for this user action.

14.74 GlobalDataSetNum

GlobalDataSetNum

This command sets a value in ’s global property set to a given numeric value.

has a global property set that lasts for the duration that is connected to the . The global property set consists of several named properties that can all be accessed by any user action. This allows for sharing of information between different actions, or for an action to persist information between one execution and another.

Syntax:

GlobalDataSetNum(PropertyName, Value)

Parameters:

- **PropertyName**: This string value represents the name of an individual property that you want to set to the given value. If the given property does not exist, it is automatically created.

- **Value**: The numerical value to store in the global property set.

  This value will remain in the global property set until loses connection with the , or until it is overwritten with a call to **GlobalDataSetNum** or **GlobalDataSetStr**.

Example:

```
' Store the contents of [Data1] in property “LastValue”.
GlobalDatSetNum(“LastValue”, [Data1])
```

Notes:

- You can also use the **LocalDataSetNum** command to set data in the local property set. The local property set only contains properties for the current action while it is executing, and can be useful for extending storage beyond the 11 [**Data*n***] macro variables.

- You can use the **GlobalDataSetStr** command to store a string value in a property.

14.75 GlobalDataSetStr

GlobalDataSetStr

This command sets a value in ’s global property set to a given string value.

has a global property set that lasts for the duration that is connected to the . The global property set consists of several named properties that can all be accessed by any user action. This allows for sharing of information between different actions, or for an action to persist information between one execution and another.

Syntax:

GlobalDataSetStr(PropertyName, Text)

Parameters:

- **PropertyName**: This string value represents the name of an individual property that you want to set to the given value. If the given property does not exist, it is automatically created.

- **Text**: The string value to store in the global property set.

  This value remains in the global property set until loses connection with the , or until it is overwritten with a call to **GlobalDataSetNum** or **GlobalDataSetStr**.

Example:

```
' Store the current account code in property “OldAccountCode”.
GlobalDatSetStr(“OldAccountCode”, [AccountCode])
```

Notes:

- You can also use the **LocalDataSetStr** command to set data in the local property set. The local property set only contains properties for the current action while it is executing, and can be useful for extending storage beyond the 11 [**Data*n***] macro variables.

- You can use the **GlobalDataSetNum** command to store a numerical value in a property.

14.76 Gosub...Return

Gosub...Return

This command sequence provides the ability to branch the code to perform a subroutine, before returning to the calling code to continue execution.

A subroutine is a small piece of code that does some task that is likely to be performed very often, or in multiple places. Rather than include the same piece of code in several locations, the code is written as a subroutine, and then a single line calls the subroutine when and where needed.

Syntax:

Gosub(LineLabel)

Parameter:

LineLabel: The line label that denotes the first line of code for the subroutine. When the execution reaches the **Gosub** command, it will jump to this line label.

A line label is case insensitive, but must contain no spaces or punctuation.

Example:

```
' Call the Subroutine that activates and restores
' the Notepad application if present, or runs it if
' it is not open yet.
  Gosub(“ActivateNotepad”)
    .
    .
' Sub routine that runs or activates/restores
' the Notepad application.
ActivateNotepad:
  AppActivateLikeShell(“Untitled - Notepad”, “NOTEPAD.EXE”)
  YieldToOs
' Return execution to line following
' original Gosub macro statement.
  Return
```

Notes:

- Although you can use **Gosub** to a specific line label as many times as you want, there can only be one instance of that line label in the user action, i.e., the subroutine defined by a line label should occur only once in the action.

- You can call one subroutine from another. However there is a limit of 15 “nested” calls, i.e., where one subroutine calls another subroutine. In a similar fashion, a subroutine could call itself, as long as there are no more than 15 of these “nested” subroutine calls.

14.77 Goto

Goto

This command continues execution of the user action at the given line label.

Syntax:

Goto(LineLabel)

Parameters:

LineLabel: The line label that denotes the first line of code that the user action should jump to before continuing execution. A line label is case insensitive, but must contain no spaces or punctuation.

Example:

```
' Temporarily ignore macro commands
' following the Goto statement.
Goto(“Exit”)
  .
  .
Exit:
```

Notes:

- You can use this command to jump backwards as well as forwards in a user action.
- Although you can use **Goto*XXX*** commands to go to a specific line label as many times as you want, there can only be one instance of that line label in the user action, i.e., the subroutine defined by a line label should occur only once in the action.

14.78 GotoIfAppActive

GotoIfAppActive

This command continues execution of a user action at a given line label if an open application window has the leftmost part of its titlebar text match a specified string. If the given window cannot be located, then execution carries on at the next line of the script.

Syntax:

GotoIfAppActive(LineLabel, WindowTitle, Condition)

Parameters:

- **LineLabel**: The line label that denotes the first line of code that the user action should jump to before continuing execution, if the given window title can be found.

  A line label is case insensitive, but must contain no spaces or punctuation.

- **WindowTitle**: The left part of the title of the application window to find.

  The name that appears in the titlebar need not be fully specified. For instance, “Calc” would still find an open application with titlebar text “Calculator”. The comparison is also not case sensitive; i.e., “Calculator” and “calculator” appear identical.

- **Condition**: If this numerical value is 0, execution continues at the specified line label if the specified window exists.

  If this numerical value is 1, execution continues at the specified line label if the specified window does not exist.

Example:

```
' Jump to AppOpen: if Notepad is already open.
GotoIfAppActive(“AppOpen”, “Untitled - Notepad”, 0)
   .
   .
AppOpen:
```

Notes:

- This command can detect application windows that are hidden. There are many application windows that are always open under Windows but are not actually visible. Examples are the NetDDE application window (“NetDDE”) and the clipboard server application (“ClipSrv”).

- It may be more preferable to use the GotoIfAppActiveRight command instead of **GotoIfAppActive** in some circumstances. The **GotoIfAppActiveRight** command can identify an application by matching the right portion of its titlebar text instead of the left part.

- Although you can use **Goto*XXX*** commands to go to a specific line label as many times as you want, there can only be one instance of that line label in the user action, i.e., the subroutine defined by a line label should occur only once in the action.

14.79 GotoIfAppActiveChild

GotoIfAppActiveChild

This command continues execution of a user action at a given line label if a given application window has a particular child window open. Both the application and child windows are located based on a match with the leftmost part of their titlebar text. If the given window cannot be located, execution carries on at the next line of the script.

Syntax:

GotoIfAppActiveChild(LineLable, AppTitle, ChildTitle, Condition)

Parameters:

- **LineLabel**: The line label that denotes the first line of code that the user action should jump to before continuing execution, if the given child window title can be found.

  A line label is case insensitive, but must contain no spaces or punctuation.

- **AppTitle**: The left part of the title of the application window to find.

  The name that appears in the titlebar need not be fully specified. For instance, “Calc” would still find an open application with titlebar text “Calculator”. The comparison is also not case sensitive; i.e., “Calculator” and “calculator” appear identical.

- **ChildTitle**: The left part of the title of the child window to find.

  The name that appears in the titlebar need not be fully specified. For instance, “Main” would still find an open child window with titlebar text “Main Menu”. The comparison is also not case sensitive, i.e., “Main Menu” and “main menu” appear identical.

- **Condition**: If this numerical value is 0, execution continues at the specified line label if the specified window exists.

  If this numerical value is 1, execution continues at the specified line label if the specified window does not exist.

Example:

```
' Jump to GotOrder: if the Order Form
' is open in Microsoft Access.
GotoIfAppActiveChild(“GotOrder”, “Microsoft Access”, “Order”, 0)
   .
   .
GotOrder:
```

Notes:

- This command can detect application windows that are hidden. There are many application windows that are always open under Windows but are not actually visible. Examples are the NetDDE application window (“NetDDE”) and the clipboard server application (“ClipSrv”).

- Although you can use **Goto*XXX*** commands to go to a specific line label as many times as you want, there can only be

one instance of that line label in the user action, i.e., the subroutine defined by a line label should occur only once in the action.

14.80 GotoIfAppActiveRight

GotoIfAppActiveRight

This command continues execution of a user action at a given line label if an open application window has the rightmost part of its titlebar text match a specified string. If the given window cannot be located, then execution carries on at the next line of the script.

Syntax:

GotoIfAppActiveRight(LineLabel, WindowTitle, Condition)

Parameters:

- **LineLabel**: The line label that denotes the first line of code that the user action should jump to before continuing execution, if the given window title can be found.

  A line label is case insensitive, but must contain no spaces or punctuation.

- **AppTitle**: The right part of the title of the application window to find.

  The name that appears in the titlebar need not be fully specified. For instance, “Calc” would still find an open application with titlebar text “Calculator”. The comparison is also not case sensitive; i.e., “Calculator” and “calculator” appear identical.

- **Condition**: If this numerical value is 0, execution continues at the specified line label if the specified window exists.

  If this numerical value is 1, execution continues at the specified line label if the specified window does not exist.

Example:

```
' Jump to AppOpen: if Microsoft Word is already open.
GotoIfAppActiveRight(“AppOpen”, “ Word”, 0)
   .
   .
AppOpen:
```

Notes:

- This command can detect application windows that are hidden. There are many application windows that are always open under Windows but are not actually visible. Examples are the NetDDE application window (“NetDDE”) and the clipboard server application (“ClipSrv”).

- It may be more preferable to use the GotoIfAppActive command instead of **GotoIfAppActiveRight** in some circumstances. The **GotoIfAppActive** command can identify an application by matching the left portion of its titlebar text instead of the right part.

- Although you can use **Goto*XXX*** commands to go to a specific line label as many times as you want, there can only be one instance of that line label in the user action, i.e., the subroutine defined by a line label should occur only once in the action.

14.81 GotoIfAppActiveRightChild

GotoIfAppActiveRightChild

This command continues execution of a user action at a given line label if a given application window has a particular child window open. The application window is located based on a match with the rightmost part of its titlebar text, while the child window is located based on a match with the leftmost part of its titlebar text. If the given window cannot be located, execution carries on at the next line of the script.

Syntax:

GotoIfAppActiveRightChild(LineLabel, AppTitle, ChildTitle, Condition)

Parameters:

- **LineLabel**: The line label that denotes the first line of code that the user action should jump to before continuing execution, if the given child window title can be found.

  A line label is case insensitive, but must contain no spaces or punctuation.

- **AppTitle**: The rightmost part of the title of the application window to find.

  The name that appears in the titlebar need not be fully specified. For instance, “Calc” would still find an open application with titlebar text “Calculator”. The comparison is also not case sensitive; i.e., “Calculator” and “calculator” appear identical.

- **ChildTitle**: The left part of the title of the child window to find.

  The name that appears in the titlebar need not be fully specified. For instance, “Main” would still find an open child window with titlebar text “Main Menu”. The comparison is also not case sensitive, i.e., “Main Menu” and “main menu” appear identical.

- **Condition**: If this numerical value is 0, execution continues at the specified line label if the specified window exists.

  If this numerical value is 1, execution continues at the specified line label if the specified window does not exist.

Example:

```
' Jump to ChildOpen: if Document1
' is open in Microsoft Word.
GotoIfAppActiveRightChild(“ChildOpen”, “Word”, “Document1”, 0)
   .
   .
ChildOpen:
```

Notes:

- This command can detect application windows that are hidden. There are many application windows that are always open under Windows but are not actually visible. Examples are the NetDDE application window (“NetDDE”) and the clipboard server application (“ClipSrv”).

- Although you can use **Goto*XXX*** commands to go to a specific line label as many times as you want, there can only be one instance of that line label in the user action, i.e., the subroutine defined by a line label should occur only once in the action.

14.82 GotoIfAppFocus

GotoIfAppFocus

This command continues execution of a user action at a given line label if the specified window, as defined by the leftmost part of its titlebar, is the currently active window. The active window is the window that currently gets keyboard input. If the given window is not the active window, execution carries on at the next line of the script.

Syntax:

GotoIfAppFocus(LineLabel, WindowTitle, Condition)

Parameters:

- **LineLabel**: The line label that denotes the first line of code that the user action should jump to before continuing execution, if the given window is the active window.

  A line label is case insensitive, but must contain no spaces or punctuation.

- **WindowTitle**: The left part of the title of the application window to check to see if it's active.

  The name that appears in the titlebar need not be fully specified. For instance, "Calc" would still find an open application with titlebar text "Calculator." The comparison is also not case sensitive; i.e., "Calculator" and "calculator" appear identical.

- **Condition**: If this numerical value is 0, execution continues at the specified line label if the specified window is the active application.

  If this numerical value is 1, execution continues at the specified line label if the specified window is not the active application.

Example:

```
' Jump to AppFocus: if Notepad is the
' active application.
GotoIfAppFocus("AppFocus", "Untitled - Notepad", 0)
   .
   .
AppFocus:
```

Notes:

- You can obtain the titlebar text for the current active application in expressions you place within macro command arguments. To do this, place the [Titlebar] macro variable within the expression.

- Although you can use **Goto*XXX*** commands to go to a specific line label as many times as you want, there can only be one instance of that line label in the user action, i.e., the subroutine defined by a line label should occur only once in the action.

14.83 GotoIfAppFocusChild

GotoIfAppFocusChild

This command continues execution of a user action at a given line label if the specified child window of a given application is the currently active window. The active window is the window that currently gets keyboard input. If the given window is not the active window then execution carries on at the next line of the script.

Both the application window and child windows are identified using the leftmost part of their titlebar text.

Syntax:

GotoIfAppFocusChild(LineLabel, WindowTitle, ChildWindowTitle, Condition)

Parameters:

- **LineLabel**: The line label that denotes the first line of code that the user action should jump to before continuing execution, if the given child window is the active window.

  A line label is case insensitive, but must contain no spaces or punctuation.

- **WindowTitle**: The left part of the title of the application window that contains the child window to check.

  The name that appears in the titlebar need not be fully specified. For instance, “Calc” would still find an open application with titlebar text “Calculator”. The comparison is also not case sensitive; i.e., “Calculator” and “calculator” appear identical.

- **ChildWindowTitle**: The left part of the title of the child window to check to see if it is active.

- **Condition**: If this numerical value is 0, execution continues at the specified line label if the specified child window is the active window, i.e., receives keyboard input.

  If this numerical value is 1, execution continues at the specified line label if the specified child window is not the active window.

Example:

```
' Jump to GotOrders: if the Orders Form is
' the active child window Microsoft Access.
GotoIfAppFocusChild(“GotOrders”, “Microsoft Access”, “Orders”, 0)
   .
   .
GotOrders:
```

Notes:

Although you can use **Goto*XXX*** commands to go to a specific line label as many times as you want, there can only be one instance of that line label in the user action, i.e., the subroutine defined by a line label should occur only once in the action.

14.84 GotoIfAppFocusRight

GotoIfAppFocusRight

This command continues execution of a user action at a given line label if the specified window, as defined by the rightmost part of its titlebar, is the currently active window. The active window is the window that currently gets keyboard input. If the given window is not the active window, execution carries on at the next line of the script.

Syntax:

GotoIfAppFocusRight(LineLabel, WindowTitle, Condition)

Parameters:

- **LineLabel**: The line label that denotes the first line of code that the user action should jump to before continuing execution, if the given child window is the active window.

  A line label is case insensitive, but must contain no spaces or punctuation.

- **WindowTitle**: The right part of the title of the application window that contains the child window to check.

  The name that appears in the titlebar need not be fully specified. For instance, “ulator” would still find an open application with titlebar text “Calculator.”The comparison is also not case sensitive; i.e., “Calculator” and “calculator” appear identical.

- **Condition**: If this numerical value is 0, execution continues at the specified line label if the specified application window is the active window, i.e., receives keyboard input.

  If this numerical value is 1, execution continues at the specified line label if the specified application window is not the active window.

Example:

```
' Jump to AppFocus: if Microsoft Word is the
' active application.
GotoIfAppFocusRight(“AppFocus”, “Word”, 0)
   .
   .
AppFocus:
```

Notes:

- You can obtain the titlebar text for the current active application in expressions you place within macro command arguments. To do this, place the [Titlebar] macro variable within the expression.

- Although you can use **Goto*XXX*** commands to go to a specific line label as many times as you want, there can only be one instance of that line label in the user action, i.e., the subroutine defined by a line label should occur only once in the action.

14.85 GotoIfAppFocusRightChild

GotoIfAppFocusRightChild

This command continues execution of a user action at a given line label if the specified child window of a given application is the currently active window. The active window is the window that currently gets keyboard input. If the given window is not the active window, execution carries on at the next line of the script.

The application window is identified using the rightmost part of its titlebar text, while the child window is identified using the leftmost part of its titlebar text.

Syntax:

GotoIfAppFocusRightChild(LineLabel, WindowTitle, ChildWindowTitle, Condition)

Parameters:

- **LineLabel**: The line label that denotes the first line of code that the user action should jump to before continuing execution, if the given child window is the active window.

  A line label is case insensitive, but must contain no spaces or punctuation.

- **WindowTitle**: The right part of the title of the application window that contains the child window to check.

  The name that appears in the titlebar need not be fully specified. For instance, “ulator” would still find an open application with titlebar text “Calculator”. The comparison is also not case sensitive; i.e., “Calculator” and “calculator” appear identical.

- **ChildWindowTitle**: The left part of the title of the child window to check to see if it’s active.

- **Condition**: If this numerical value is 0, then execution continues at the specified line label if the specified child window is the active window, i.e., receives keyboard input.

  If this numerical value is 1, execution continues at the specified line label if the specified child window is not the active window.

Example:

```
' Jump to ChildActive: if Document1 is the
' active child window in Microsoft Word.
GotoIfAppFocusRightChild(“ChildActive”, “Word”, “Document1”, 0)
   .
   .
ChildActive:
```

Notes:

Although you can use **Goto*XXX*** commands to go to a specific line label as many times as you want, there can only be one instance of that line label in the user action, i.e., the subroutine defined by a line label should occur only once in the action.

14.86 GotoIfCallType

GotoIfCallType

This command continues execution of a user action at a given line label if the current call is of a particular type. The current call is the call selected in the active call list if the user action fires via a hot key or button, but if the action fires via a rule, then the current call is the call that caused the rule to fire. If the current call is not of the correct type, then execution carries on at the next line of the script.

Syntax:

GotoIfCallType(LineLabel, CallType, Condition)

Parameters:

- **LineLabel**: The line label that denotes the first line of code that the user action should jump to before continuing execution, if the current call is of the type depicted by the “CallType” parameter.

  A line label is case insensitive, but must contain no spaces or punctuation.

- **CallType**: A numerical value that depicts the type of call to check for, as follows:

Value	Call Type
0	Internal call
1	External (trunk line) call
2	Outbound call
3	Inbound call
4	Held call
5	Unheld call
6	Answered call
7	Unanswered call (alerting)
8	The number or received for an external call was against the ‘s telephone data import.
9	The number or received for an external call was not against the ‘s telephone data import. If a call is received without , it will also match this call type.
10	was received for an inbound call (external calls only).
11	was not received for an inbound call (external calls only).

- **Condition**: If this numerical value is 0, execution continues at the specified line label if the current call matches the specified call type.

  If this numerical value is 1, execution continues at the specified line label if the current call does not match the specified call type.

Example:

```
' Jump to : if was received
' for the current call.
```

```
GotoIfCallType(" ", 10, 0)
    .
    .
:
```

Notes:

- You can force selection of a call within 's call list by using the **CallSelect** command. You usually need to use the **CallSelect** command when you have written a user action that makes a new call and you need to refer to information relating to the new call in the same action (for example, by using macro variables or commands like the **GotoIfCallType** command).

- Although you can use **Goto*XXX*** commands to go to a specific line label as many times as you want, there can only be one instance of that line label in the user action, i.e., the subroutine defined by a line label should occur only once in the action.

14.87 GotoIfDateBetween

GotoIfDateBetween

This command continues execution of a user action at a given line label if a given date is between two specific dates. If the given date is not in the specified range, execution carries on at the next line of the script.

Syntax:

GotoIfDateBetween(LineLabel, Date1, Date2, Date, IgnoreBadDates)

Parameters:

- **LineLabel**: The line label that denotes the first line of code that the user action should jump to before continuing execution, if the given date is between the specified range.

  A line label is case insensitive, but must contain no spaces or punctuation.

- **Date1**: This string value represents the lower part of the date range to compare against. The date should be written using the format specified in Windows Regional Settings, e.g., in the this would be “ .”

- **Date2**: This string value represents the upper part of the date range to compare against. The date should be written using the format specified in Windows Regional Settings, e.g., in the this would be “ .”

- **Date**: This string value represents the date that is checked to ensure it is between “Date1” and “Date2”. The date should be written using the format specified in Windows Regional Settings, e.g., in the this would be “ .”

  If this value is between “Date1” and “Date2,” execution of the user action will continue at the specified line label, otherwise it will continue on the next line.

- **IgnoreBadDates**: If this numerical value is 0, and any of the specified dates is invalid, an error is generated and execution of the user action halts.

  If the value is 1, then bad dates are ignored, and execution of the user action continues on the next line of the script, as if the date specified was not within the given range.

  An example of a bad date could be “ ” (February only has 28 or 29 days), or a date that cannot be interpreted, e.g., “”.

Example:

```
' Jump to IsBefore2006: if the current
' date is before the year 2006.
GotoIfDateBetween(“IsBefore2006”, “1/1/1900”, “ /2005”, [LongDate], 0)
   .
   .
IsBefore2006:
```

Notes:

- The date to be compared can usually be best retrieved using the [ShortDate] or [LongDate] macro variables, which return today’s date based on the current date on the computer that the user action is executing on.

- Dates can be entered in different styles, e.g., “ ”, “ ”, etc.

- When you specify date arguments in this command, it is advisable to use a long date format (, e.g.,) because the year part is specified explicitly. Other date formats do not explicitly specify the year of a date. For instance, the dates and in short date format would both be returned from Windows as “ ” when you used the [**ShortDate**] macro variable.

- Although you can use **Goto*XXX*** commands to go to a specific line label as many times as you want, there can only be one instance of that line label in the user action, i.e., the subroutine defined by a line label should occur only once in the action.

14.88 GotoIfDDESendCmd

GotoIfDDESendCmd

This command continues execution of a user action at a given line label if the given DDE command fails. If the DDE command succeeds, execution carries on at the next line of the script.

Syntax:

GotoIfDDESendCmd(LineLabel, Channel, CommandString, IgnoreMainErrors, Condition)

Parameters:

- **LineLabel**: The line label that denotes the first line of code that the user action should jump to before continuing execution, if the given DDE fails.

  A line label is case insensitive, but must contain no spaces or punctuation.

- **Channel**: The channel number of the DDE conversation to send a command to. This can be a value from 1 to 6.

- **CommandString**: The application-specific command to send to the DDE server

- **IgnoreMainErrors**: If this numerical value is 0, execution of the user action stops with an error if a fatal DDE error occurs.

  If the value is 1, a fatal DDE error is considered as the DDE server failing the DDE command, and so is used in the branching logic.

  A fatal DDE error can include such things as:

  - The command times out.
  - No conversation is active on the given channel.
  - The command string is empty.

- **Condition**: If this numerical value is 0 then execution of the user action continues at the specified line label if the DDE command fails.

  If the value is 1, execution continues at the line label if the DDE command succeeds.

Example:

```
' Jump to DDEFailed: if Microsoft Access rejects
' the DDE find record command.
GotoIfDDESendCmd("DDEFailed", 1, "[FindRecord ""
    .
    .
DDEFailed:
```

Notes:

Although you can use **Goto*XXX*** commands to go to a specific line label as many times as you want, there can only be one instance of that line label in the user action, i.e., the subroutine defined by a line label should occur only once in the action.

14.89 GotoIfFileExists

GotoIfFileExists

This command continues execution of a user action at a given line label if the specified file name exists. If the file does not exist, execution carries on at the next line of the script.

Syntax:

GotoIfFileExists(LineLabel, Filename, Condition)

Parameters:

- **LineLabel**: The line label that denotes the first line of code that the user action should jump to before continuing execution, if the given file exists.

  A line label is case insensitive, but must contain no spaces or punctuation.

- **Filename**: The fully pathed filename to the file that is being checked.

- **Condition**: If this numerical value is 0, execution continues at the specified line label if the given file exists.

  If the value is 1, execution continues at the line label if the given file does not exist.

Example:

```
' Jump to MSWExists: if file "C:\WINDOWS\WINWORD.INI" exists.
GotoIfFileExists("MSWExists", "C:\WINDOWS\WINWORD.INI", 0)
   .
   .
MSWExists:
```

Notes:

- Remember that the location of some files are not always the same between different computers, depending on how software has been installed. It is often better if, where possible, you use macro variables such as [**WinDir**] to define the path of the file, and only hardcode the filename rather than the entire path, e.g., [WinDir] + "WIN.INI", rather than "C:\WINDOWS\WIN.INI".

- Although you can use **Goto*XXX*** commands to go to a specific line label as many times as you want, there can only be one instance of that line label in the user action, i.e., the subroutine defined by a line label should occur only once in the action.

14.90 GotoIfMessageBox

GotoIfMessageBox

This command continues execution of a user action at a given line label if the user clicks a particular button in the displayed message box. If the given button is not selected, execution carries on at the next line of the script.

Syntax:

GotoIfMessageBox(LineLabel, Message, ButtonType, Title, ButtonToClick, Condition)

Parameters:

- **LineLabel**: The line label that denotes the first line of code that the user action should jump to before continuing execution, if the given button is clicked in the displayed message box.

  A line label is case insensitive, but must contain no spaces or punctuation.

- **Message**: The message to display to the user in the message box.

- **ButtonType**: This numerical value indicates the buttons that should be available in the message box. It can also be used to identify the type of message box to the user through an icon, as well identifying which button should be the default.

  The buttons to display are indicated by one of the following values:

Value	Button
0	Displays **OK** button only.
1	Displays **OK** and **Cancel** buttons.
2	Displays **Abort**, **Retry**, and **Ignore** buttons.
3	Displays **Yes**, **No**, and **Cancel** buttons.
4	Displays **Yes** and **No** buttons.
5	Displays **Retry** and **Cancel** buttons.

  If you want to display an icon on the message box, add one of the following values to the number.

Value	Icon
16	Displays a Stop icon.
32	Displays a Question Mark icon.
48	Displays an Exclamation Mark icon.
64	Displays an Information icon.

  If you want to assert the default button, add one of the following values:

Value	Default Button
0	First button is default.

256	Second button is default.
512	Third button is default.

For example, specifying “4 + 32 + 256” would display **Yes** and **No** buttons on the message box (4), with a question mark icon (32), and the second button would be the default button (256).

- **Title**: The text to display in the titlebar of the message box.

- **ButtonToClick**: This numerical value defines the button that, if clicked, will cause the command to jump to the specified line label. It can be one of the following values:

Value	Default Button
1	OK
2	Cancel
3	Abort
4	Retry
5	Ignore
6	Yes
7	No

- **Condition**: This numerical value defines how the button that the user clicked is compared with the “ButtonToClick” parameter to decide whether the command subsequently jumps to the given line label. The value can be one of the following:

Value	Action
0	Execution continues at the specified line label if the number representing the button clicked match the “ButtonToClick” parameter
1	Execution continues at the specified line label if the number representing the button clicked does match the “ButtonToClick” parameter.
2	Execution continues at the specified line label if the number representing the button clicked is hig than the “ButtonToClick” parameter.
3	Execution continues at the specified line label if the number representing the button clicked is hig or equal to the “ButtonToClick” parameter.
4	Execution continues at the specified line label if the number representing the button clicked is low than the “ButtonToClick” parameter.
5	Execution continues at the specified line label if the number representing the button clicked is low equal to the “ButtonToClick” parameter

Example:

```
' Jump to UserYes: if the user clicked the Yes button.
GotoIfMessageBox(“UserYes”, “Are you an existing customer?” + 10 + “(Yes Or No)”, 4+32, “”, 6,  0)
```

.

.

```
UserYes:
```

Notes:

- The longest message string is 1024 characters. The message string will be truncated after 1024 characters. However, a message string with more than 255 characters without an intervening space will be truncated after 255 characters.
- You can insert line breaks into the message string by inserting ASCII character 10 into the string, e.g.,
- “This is line 1 ” + {10} + “This is line 2”.
- Although you can use **Goto*XXX*** commands to go to a specific line label as many times as you want, there can only be one instance of that line label in the user action, i.e., the subroutine defined by a line label should occur only once in the action.

14.91 GotoIfMessageBoxCustom

GotoIfMessageBoxCustom

Prior to version 4, this command was an alternative to the **GotoIfMessageBox** command, but with a different style of message box. In version 4, the message boxes displayed are identical, and as such the commands provide the same functionality.

14.92 GotoIfNoCalls

GotoIfNoCalls

This command continues execution of a user action at a given line label if the number of calls that are currently active at the extension associated with matches the specified condition. If the number of calls does not match the condition then execution carries on at the next line of the script.

Syntax:

GotoIfNoCalls(LineLabel, NumberCalls, Condition)

Parameters:

- **LineLabel**: The line label that denotes the first line of code that the user action should jump to before continuing execution, if the number of calls at the extension associated with matches the “Condition” parameter.

  A line label is case insensitive, but must contain no spaces or punctuation.

- **NumberCalls**: This numerical value defines the number of calls to compare against the actual number of calls active at the extension.

- **Condition**: This value defines how “NumberCalls” is compared against the actual number of calls active at the extension. The value is interpreted as follows:

Value	Description
0	Continues execution at the given line label if the number of active calls matches the “NumberCalls” parameter.
1	Continues execution at the given line label if the number of active calls does not match the “NumberC parameter.
2	Continues execution at the given line label if the number of active calls is higher than the “NumberCa parameter.
3	Continues execution at the given line label if the number of active calls is higher or equal to the “NumberCalls” parameter.
4	Continues execution at the given line label if the number of active calls is lower than the “NumberCal parameter.
5	Continues execution at the given line label if the number of active calls is lower or equal to the “NumberCalls” parameter.

Example:

```
' Jump to 2Calls: if there are 2 calls in the call list.
GotoIfNoCalls(“2Calls”, 2, 0)
   .
   .
2Calls:
```

Notes:

- You can also obtain the current number of active calls in expressions you use in macro commands using the [**Calls**] macro variable.

- Although you can use **Goto*XXX*** commands to go to a specific line label as many times as you want, there can only be one instance of that line label in the user action, i.e., the subroutine defined by a line label should occur only once in the action.

14.93 GotoIfNoRecords

GotoIfNoRecords

This command is used to branch execution of the macro based on whether the query performed returned any records.

Syntax:

GotoIfNoRecords(LineLabel, Channel, Condition)

Parameters:

- **LineLabel**: The line label that denotes the first line of code that the user action should jump to before continuing execution, if the query result matches the "Condition" parameter.
  A line label is case insensitive, but must contain no spaces or punctuation.

- **Channel**: The channel number of the ODBC connection to check. This can be a value from 1 to 3.

- **Condition**: If this value is 0, and the query has returned no records, then macro execution will continue at the LineLabel.
  If this value is 1, and the query has returned records, then macro execution will continue at the LineLabel.

Example:

```
' Jump to the "NoRecs" Label if the resultant query
' returned no results.
GotoIfNoRecords("NoRecs", 1, 0, 0)
```

Notes:

By default, errors generated will stop execution of the macro. You can switch off this functionality with the SetErrorsFatal command. Having done so, the error is available using the [ErrorDesc] and [ErrorNum] macro variables.

14.94 GotoIfNumValue

GotoIfNumValue

This command continues execution of a user action at a given line label based on the comparison between two numeric parameters. If the comparison is negative then execution carries on at the next line of the script.

Syntax:

GotoIfNumValue(LineLabel, Number1, Number2, Condition)

Parameters:

- **LineLabel**: The line label that denotes the first line of code that the user action should jump to before continuing execution, based on the comparison between the “Number1” and “Number2” parameters.

  A line label is case insensitive, but must contain no spaces or punctuation.

- **Number1**: This is the first number that is compared against “Number2” based on the “Condition” parameter.

- **Number2**: This is the number that is compared against “Number1” based on the “Condition” parameter.

- **Condition**: This value defines how “Number1” is compared against “Number2”. The value is interpreted as follows:

Value	Action
0	Continues execution at the given line label if “Number1” equals “Number2.”
1	Continues execution at the given line label if “Number1” does not equal “Number2.”
2	Continues execution at the given line label if “Number1” is higher than “Number2.”
3	Continues execution at the given line label if “Number1” is higher or equal to “Number2.”
4	Continues execution at the given line label if “Number1” is lower to “Number2.”
5	Continues execution at the given line label if “Number1” is lower or equal to “Number2.”

Example:

```
' Jump to 700Plus: if the current call
' is on a line higher or equal to 700.
GotoIfNumValue(“700Plus”, [Line], 700, 3)
   .
   .
700Plus:
```

Notes:

Although you can use **Goto*XXX*** commands to go to a specific line label as many times as you want, there can only be one instance of that line label in the user action, i.e., the subroutine defined by a line label should occur only once in the action.

14.95 GotoIfStrLen

GotoIfStrLen

This command continues execution of a user action at a given line label based on the comparison of the length of the given string and the provided comparison length. If the comparison is negative, execution carries on at the next line of the script.

Syntax:

GotoIfStrLen(LineLabel, String, StringLength, Condition)

Parameters:

- **LineLabel**: The line label that denotes the first line of code that the user action should jump to before continuing execution, based on the comparison between the length of string “String” and the length value provided in “StringLength.”

  A line label is case insensitive, but must contain no spaces or punctuation.

- **String**: The string whose length is compared against the “StringLength” parameter.

- **StringLength**: A numerical value that defines a string length that is compared against the length of the string in “String.”

- Condition: This value defines how the length of “String” is compared against the “StringLength” value. The value is interpreted as follows:

Value	Action
0	Continues execution at the given line label if the length of “String” equals “StringLength.”
1	Continues execution at the given line label if the length of “String” does not equal “StringLength.”
2	Continues execution at the given line label if the length of “String” is higher than “StringLength.”
3	Continues execution at the given line label if the length of “String” is higher or equal to “StringLength.”
4	Continues execution at the given line label if the length of “String” is lower than “StringLength.”
5	Continues execution at the given line label if the length of “String” is lower or equal to “StringLength.”

Example:

```
' Jump to DDILen4: if 4 digit DID is being
' received from the network provider.
GotoIfStrLen(“DDILen4”, [DDIDigits], 4, 0)
   .
   .
DDILen4:
```

Notes:

Although you can use **Goto*XXX*** commands to go to a specific line label as many times as you want, there can only be one instance of that line label in the user action, i.e., the subroutine defined by a line label should occur only once in the action.

14.96 GotoIfStrValue

GotoIfStrValue

This command continues execution of a user action at a given line label based on the comparison of the two strings. If the comparison is negative, execution carries on at the next line of the script.

Syntax:

GotoIfStrValue(LineLabel, String1, String2, Condition)

Parameters:

- **LineLabel**: The line label that denotes the first line of code that the user action should jump to before continuing execution, based on the comparison between the “String1” and “String2.”

  A line label is case insensitive, but must contain no spaces or punctuation.

- **String1**: The string to compare against “String2.”

- **String2**: The string to compare against “String1.”

- **Condition**: This value defines how the “String1” is compared against “String2.”

Value	Action
0	Continues execution at the given line label if “String1” is equal to “String2.”
1	Continues execution at the given line label if “String1” does not equal “String2."
2	Continues execution at the given line label if “String1” is higher than “String2.”
3	Continues execution at the given line label if “String1” is higher or equal to “String2."
4	Continues execution at the given line label if “String1” is lower than “String2.”
5	Continues execution at the given line label if “String1” is lower or equal to “String2.”

Example:

```
' Jump to 14809619000 : if the
' current call is from/to 14809619000 .
GotoIfStrValue(“ 14809619000 ”, [Digits], “ 14809619000 ”, 0)
   .
   .
14809619000:
```

Notes:

- When comparing strings, a string is considering higher than another string at the first instance of a character that is later in the alphabet, i.e., “B” is higher than “A”. If one string is shorter than the other, then the shorter string is considered ‘lower’ than the other string as long as it matches up to the end of the string.

- For example, “Apple Mac” is considered higher than “Apple” because it is identical in the first 5 letters, but then has more characters, so its length is longer, and so it is a higher value.

- String comparisons are also case insensitive, so “Apple” equals “APPLE”.

- Although you can use **Goto*XXX*** commands to go to a specific line label as many times as you want, there can only be one instance of that line label in the user action, i.e., the subroutine defined by a line label should occur only once in the action.

14.97 GotoIfStrValueLeft

GotoIfStrValueLeft

This command continues execution of a user action at a given line label based on the comparison of the given left part of two strings. If the comparison is negative then execution carries on at the next line of the script.

This command is useful for comparing area codes of two telephone numbers without having to remove the non-area code part of both telephone numbers first.

Syntax:

GotoIfStrValueLeft(LineLabel, String1, String2, NumberChars, Condition)

Parameters:

- **LineLabel**: The line label that denotes the first line of code that the user action should jump to before continuing execution, based on the comparison between the “String1” and “String2.”

  A line label is case insensitive, but must contain no spaces or punctuation.

- **String1**: The string to compare against “String2.”

- **String2**: The string to compare against “String1.”

- **NumberChars**: The number of characters to compare in both strings, starting from the leftmost character.

- **Condition**: This value defines how the “String1” is compared against “String2.”

Value	Action
0	Continues execution at the given line label if the first “NumberChars” characters of “String1” is equal to same number of characters in “String2.”
1	Continues execution at the given line label if the first “NumberChars” characters of “String1” does not eq the same number of characters in “String2.”
2	Continues execution at the given line label if the first “NumberChars” characters of “String1” is higher th the same number of characters in “String2.”
3	Continues execution at the given line label if the first “NumberChars” characters of “.Sring1” is higher or equal to the same number of characters in “String2.”
4	Continues execution at the given line label if the first “NumberChars” characters of “String1” is lower tha same number of characters in “String2.”
5	Continues execution at the given line label if the first “NumberChars” characters of “String1” is lower or to the same number of characters in “String2.”

Example:

```
' Jump to 1480 : if the current
' call is from/to local area code 1480 .
GotoIfStrValueLeft(“ 1480 ”, [Digits], “ 1480 ”, 4 , 0)
```

```
.
.
1480:
```

Notes:

- When comparing strings, a string is considering higher than another string at the first instance of a character that is later in the alphabet, i.e., “B” is higher than “A.” If one string is shorter than the other, then the shorter string is considered ‘lower’ than the other string as long as it matches up to the end of the string.

  For example, “Apple Mac” is considered higher than “Apple” because it is identical in the first five letters, but then has more characters, so its length is longer, and so it is a higher value.

- String comparisons are also case insensitive, so “Apple” equals “APPLE.”

- Although you can use **Goto*XXX*** commands to go to a specific line label as many times as you want, there can only be one instance of that line label in the user action, i.e., the subroutine defined by a line label should occur only once in the action.

14.98 GotoIfStrValueLike

GotoIfStrValueLike

This command continues execution of a user action at a given line label if one string expression matches another, using a “wildcard” match. If the comparison is negative then execution carries on at the next line of the script.

A “wildcard” match allows for certain characters to represent one or more of any character. This is useful for checking a string to see if it contains certain text.

Syntax:

GotoIfStrValueLike(LineLabel, String1, String1, Condition)

Parameters:

- **LineLabel**: The line label that denotes the first line of code that the user action should jump to before continuing execution, based on the comparison between the “String1” and “String2.”

  A line label is case insensitive, but must contain no spaces or punctuation.

- **String1**: The string to compare against “String2.”

- **String2**: The string to compare against “String1.” This string can contain special characters performed in the wildcard match, as follows:

Character	Description
*	Represents zero or more characters
?	Represents a single character
#	Represents a single numeric digit
[charlist]	Represents any character contained within the brackets, e.g., “[A-Z]” matches any uppercase letter while “[A-Za-z0-9]” will match any alphanumeric character.
[!charlist]	Represents any character not contained within the brackets, e.g., “[!0-9]” matches any non-numeric character.

- **Condition**: If this numerical value is 0, then execution continues at the specified line label if “String1” pattern matches the string expression “String2.”

  If this value is 1, then execution continues at the specified line label if “String1” does not pattern match string expression “String2.”

Examples:

```
' Jump to SalesLine: if the current call
' was received on any of the sales lines.
GotoIfStrValueLike(“SalesLine”, [DNIS], “*Sales*”, 0)
   .
   .
SalesLine:
```

The following table gives examples of “String1” and “String2,” and whether they would be considered a match.

String1	String2	Matches?
aBba	a*a	Yes
aBba	a?a	No
a*a	a[*]a	Yes
aBa	a[*]a	No
1 800 277537	1 800 ######	Yes
1 800 APPLES	1 800 ######	No
A test	[A-Z]*	Yes
a test	[A-Z]*	No

Notes:

- If you need to insert one of the special characters into “String2” without it being interpreted as a special character, then wrap the character in square brackets, e.g., [*]. The third example in “More Examples” is a demonstration of this.

- Although you can use **Goto*XXX*** commands to go to a specific line label as many times as you want, there can only be one instance of that line label in the user action, i.e., the subroutine defined by a line label should occur only once in the action.

14.99 GotoIfStrValueMid

GotoIfStrValueMid

This command continues execution of a user action at a given line label if a subsection of one string matches the subsection of another string. If the comparison is negative then execution carries on at the next line of the script.

Syntax:

GotoIfStrValueMid(LineLabel, String1, String2, StartCharacter, NumberChars, Condition)

Parameters:

- **LineLabel**: The line label that denotes the first line of code that the user action should jump to before continuing execution, based on the comparison between the “String1” and “String2.”

  A line label is case insensitive, but must contain no spaces or punctuation.

- **String1**: The string to compare against “String2."

- **String2**: The string to compare against “String1.”

- **StartCharacter**: The first character to begin the comparison at, where “1” represents the first character in both string.

- **NumberChars**: The number of characters to compare in both strings.

- **Condition**: This value defines how the “String1” is compared against “String2.”

Value	Action
0	Continues execution at the given line label if “String1” is equal to “String2.”
1	Continues execution at the given line label if “String1” does not equal “String2.”
2	Continues execution at the given line label if “String1” is higher than “String2.”
3	Continues execution at the given line label if “String1” is higher or equal to “String2.”
4	Continues execution at the given line label if “String1” is lower than “String2.”
5	Continues execution at the given line label if “String1” is lower or equal to “String2.”

Example:

```
' Jump to Is213: if the middle of the [Data1] variable
' contains 213.
GotoIfStrValueMid(“Ext213”, [Data1], “213”, 5, 3, 0)
   .
   .
Is213:
```

Notes:

- When comparing strings, a string is considering higher than another string at the first instance of a character that is later in the alphabet, i.e., “B” is higher than “A”. If one string is shorter than the other, then the shorter string is considered ‘lower’ than the other string as long as it matches up to the end of the string.

  For example, “Apple Mac” is considered higher than “Apple” because it is identical in the first five letters, but then has more characters, so its length is longer, and so it is a higher value.

- String comparisons are also case insensitive, so “Apple” equals “APPLE.”

- Although you can use **Goto*XXX*** commands to go to a specific line label as many times as you want, there can only be one instance of that line label in the user action, i.e., the subroutine defined by a line label should occur only once in the action.

14.100 GotoIfStrValueRight

GotoIfStrValueLeft

This command continues execution of a user action at a given line label if the rightmost part of two strings match. If the comparison is negative then execution carries on at the next line of the script.

This command is quicker than using the DataSetStrRight command to concatenate both strings to the appropriate length before comparing, as well as not affecting the strings being compared.

Syntax:

GotoIfStrValueRight(LineLabel, String1, String2, NumberChars, Condition)

Parameters:

- **LineLabel:** The line label that denotes the first line of code that the user action should jump to before continuing execution, based on the comparison between the “String1” and “String2”.

  A line label is case insensitive, but must contain no spaces or punctuation.

- **String1:** The string to compare against “String2.”
- **String2:** The string to compare against “String1.”
- **NumberChars:** The rightmost number of characters to compare in both strings.
- **Condition:** This value defines how the “String1” is compared against “String2.”

Value	Action
0	Continues execution at the given line label if “String1” is equal to “String2.”
1	Continues execution at the given line label if “String1” does not equal “String2.”
2	Continues execution at the given line label if “String1” is higher than “String2.”
3	Continues execution at the given line label if “String1” is higher or equal to “String2.”
4	Continues execution at the given line label if “String1” is lower than “String2.”
5	Continues execution at the given line label if “String1” is lower or equal to “String2.”

Example:

```
' Jump to LineEnd10: if the current call
' is on a line than ends with digits 10.
GotoIfStrValueRight(“LineEnd10”, [Line], “10”, 2, 0)
   .
   .
LineEnd10:
```

Notes:

- When comparing strings, a string is considering higher than another string at the first instance of a character that is later in the alphabet, i.e., “B” is higher than “A”. If one string is shorter than the other, then the shorter string is considered ‘lower’ than the other string as long as it matches up to the end of the string.

  For example, “Apple Mac” is considered higher than “Apple” because it is identical in the first 5 letters, but then has more characters, so its length is longer, and so it is a higher value.

- String comparisons are also case insensitive, so “Apple” equals equal “APPLE”.

- Although you can use GotoXXX commands to go to a specific line label as many times as you want, there can only be one instance of that line label in the user action, i.e., the subroutine defined by a line label should occur only once in the action.

14.101 GotoIfTimeBetween

GotoIfTimeBetween

This command continues execution of a user action at a given line label if a given time lies inside a specific time range. If the comparison is negative, execution carries on at the next line of the script.

Syntax:

GotoIfTimeBetween(LineLabel, Time1, Time2, Time, IgnoreBadTimes)

Parameters:

- **LineLabel**: The line label that denotes the first line of code that the user action should jump to before continuing execution, if “Time” falls between “Time1” and “Time2”.

  A line label is case insensitive, but must contain no spaces or punctuation.

- **Time1**: The lower part of the time range to compare against. This is a string expression that will be evaluated as a time based on the Windows Regional Settings. Examples include “15:30:00”, “9:30AM”, etc.

- **Time2**: The upper part of the time range to compare against. This is a string expression that will be evaluated as a time based on the Windows Regional Settings. Examples include “15:30:00”, “9:30AM”, etc.

- **Time**: The time to compare against the time range defined by “Time1” and “Time2”. This is a string expression that will be evaluated as a time based on the Windows Regional Settings. Examples include “15:30:00”, “9:30AM”, etc.

- **IgnoreBadTimes**: If this numerical value is 0, then if any of the time parameters cannot be interpreted as a time, an error will be generated, and execution will stop.

  If the value is 1, any invalid time parameters will be seen as the given time not falling into the specified range, and execution will continue on the next line of the script.

Example:

```
' Jump to NightService: if the current time
' is outside working hours.
GotoIfTimeBetween(“NightService”, “5:29 PM”, “8:59 AM”, [MediumTime], 0)
   .
   .
NightService:
```

Notes:

- The values defining the time range, “Time1” and “Time2”, do not need to be sequential, and can roll over midnight. In such an occurrence, the command will consider “Time” as being in the specified time range, if it occurs after “Time1,” or before “Time2.”

- You can obtain the time on the local computer using the [ShortTime] and [**LongTime**] macro variables.

- Although you can use **Goto*XXX*** commands to go to a specific line label as many times as you want, there can only be one instance of that line label in the user action, i.e., the subroutine defined by a line label should occur only once in the action.

14.102 GotoIfWeekDay

GotoIfWeekDay

This command continues execution of a user action at a given line label if a given date falls on a specific day of the week. If the comparison is negative, execution carries on at the next line of the script.

Syntax:

GotoIfWeekDay(LineLabel, Date, Weekday, Condition, IgnoreBadDate)

Parameters:

- **LineLabel**: The line label that denotes the first line of code that the user action should jump to before continuing execution, if the given date falls on the specific day of the week.

  A line label is case insensitive, but must contain no spaces or punctuation.

- **Date**: The date that will be compared against the given day of the week in the “Weekday” parameter. This is a string expression that is interpreted using the Windows Regional Settings.

- **Weekday**: This numerical value defines the weekday to compare against the date specified in the “Date” parameter. It can be one of the following values:

Value	Weekday
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday
7	Sunday

- **Condition**: This numerical value defines how to compare the weekday of the specified date with the “Weekday” parameter, as follows:

Value	Description
0	Continues execution at the given line label if the day of week for “Date” exactly matches the “Weekday” parameter.
1	Continues execution at the given line label if the day of week for “Date” does not match the “Weekday” parameter.
2	Continues execution at the given line label if the day of week for “Date” is higher than the “Weekday” parameter.
3	Continues execution at the given line label if the day of week for “Date” is higher or equal to the “Weekday” parameter.
4	Continues execution at the given line label if the day of week for “Date” is lower than the “Weekday” parameter.

5	Continues execution at the given line label if the day of week for “Date” is lower or equal to the “Weekday” parameter.

- **IgnoreBadDates**: If this numerical value is 0, then an invalid date in the “Date” parameter will generate an error, and execution will stop.

  If the value is 1, an invalid date in the “Date” parameter will be treated as if the day of week for “Date” did not match the “Weekday” parameter, and execution will continue on the next line of the script.

Example:

```
' Jump to IsWeekEnd: if the current date
' falls on a Saturday or Sunday.
GotoIfWeekDay(“IsWeekEnd”, , 6, 3, 0)
   .
   .
IsWeekEnd:
```

Notes:

- When you specify date arguments in this command it is advisable to use a long date format (, e.g.,) because the year part is specified explicitly. Other date formats do not explicitly specify the year of a date. For instance, the dates and in short date format would both be returned from Windows as “ ” when you used the [**ShortDate**] macro variable.

- Although you can use **Goto*XXX*** commands to go to a specific line label as many times as you want, there can only be one instance of that line label in the user action, i.e., the subroutine defined by a line label should occur only once in the action.

14.103 InputBox

InputBox

This command displays a prompt in a window for the user to enter some text. If the user subsequently clicks **OK**, the input is stored in a given [**Data*n***] macro variable.

Syntax:

InputBox(Prompt, Title, DefaultValue, DataVariable)

Parameters:

- **Prompt**: The message to display to the user to prompt the user for input.
- **Title**: The titlebar text to display in the input window.
- **DefaultValue**: The default value to display in the text area where the user enters their response.
- **DataVariable**: A numerical value between 1 and 11 that depicts which macro variable to store the results in. The number 1 will store the result in [Data1] macro variable, 2 will store the result in [Data2], and so on.

Example:

```
' Prompt the agent to enter in the service that
' the caller requires and place the answer into
' the [Data2] Macro Variable Reference.
InputBox(“What fruit do you require today?” + 10 + “Apples-1, Pears-2, Oranges-3”,“”,“3”, 2)
' End macro if nothing entered.
ExitMacroStrValue([Data2], “”, 1)
```

14.104 intAbout

intAbout

This command has been deprecated in version 4 of the CallViewer SDK.

14.105 intAutoMacro

intAutoMacro

This command displays the Rules Manager in CallViewer .

Syntax:

intAutoMacro

Parameters:

None.

Example:

```
intAutoMacro
```

Notes:

Prior to version 4, this command displayed the list of automatic macros in the CallViewer Macro Manager. Automatic macros were replaced by rules in version 4 of CallViewer .

14.106 intButtonsConfig

intButtonsConfig

This command has been deprecated in version 4 of the CallViewer SDK.

14.107 intCallDetails

intCallDetails

This command displays the Call Details window for the currently selected call.

Syntax:

intCallDetails

Parameters:

None.

Example:

```
intCallDetails
```

Notes:

You can select a particular call in the call list using the **CallSelect** command.

14.108 intClearScreen

intClearScreen

This command has been deprecated in version 4 of the CallViewer SDK.

14.109 intDebugWindow

intDebugWindow

This command displays the Simulation Window.

Syntax:

intDebugWindow

Parameters:

None.

Example:

```
intDebugWindow
```

14.110 intExit

intExit

This command quits the CallViewer application.

Syntax:

intExit

Parameters:

None.

Example:

```
intExit
```

14.111 intGWin

intGWin

This command has been deprecated in version 4 of the CallViewer SDK.

14.112 intHotkeyMgr

intHotkeyMgr

This command has been deprecated in version 4 of the CallViewer SDK.

14.113 intRefreshNetworkLink

intRefreshNetworkLink

This command has been deprecated in version 4 of the CallViewer SDK.

14.114 intSettingsCC

intSettingsCC

This command has been deprecated in version 4 of the CallViewer SDK.

14.115 intSettingsGWin

intSettingsGWin

This command has been deprecated in version 4 of the CallViewer SDK.

14.116 intSettingsAdvanced

intSettingsAdvanced

This command has been deprecated in version 4 of the CallViewer SDK.

14.117 intSettingsNetwork

intSettingsNetwork

This command has been deprecated in version 4 of the CallViewer SDK.

14.118 intSettingsWindow

intSettingsWindow

This command has been deprecated in version 4 of the CallViewer SDK.

14.119 intSizeNormal

intSizeNormal

This command maximizes the main CallViewer window.

Syntax:

intSizeNormal

Parameters:

None.

Example:

```
intSizeNormal
```

Notes:

This command has no affect if the current “Look and Feel” does not support windows that can be maximized .

14.120 intSizeSmall

intSizeSmall

This command puts the main CallViewer window into “mini” mode.

Syntax:

intSizeSmall

Parameters:

None.

Example:

```
intSizeSmall
```

Notes:

This command has no effect if the current “Look and Feel” does not support a “mini” mode.

14.121 LocalDataGet

LocalDataGet

This command retrieves a value from the user action's local property set, and stores it in a given [**Data*n***] macro variable.

Each user action has its own local property set that lasts for as long as that instance of the action is running. A local property set consists of several named properties, and can be used as a convenient extension to the [**Data*n***] macro variables.

Syntax:

LocalDataGet(PropertyName, DataVar)

Parameters:

- **PropertyName**: This string value represents the name of an individual property that you want to get the current value for. If the property name does not exist, an error is generated.

- **DataVar**: This numerical value depicts the [**Data*n***] macro variable that the property's current value will be stored in. For example, setting this value to 1 would store the property's value in the [**Data1**] macro variable.

Example:

```
' Get the macro name property from the local property set
' and store it in Data1.
LocalDataGet("MacroName", 1)
```

Notes:

- The local property set contains some default values relating to the user action that it is running in, as follows:

Property	Description
AutoMacro	If this action was executed because a rule fired, this value will be "1", otherwise it will be "0".
MacroName	The name of this action in .

- You can also use the **GlobalDataGet** command to get data from the global property set. The global property set allows for properties to be shared between running instances of user actions, and the data lasts until loses connection with .

- You use the **LocalDataSetStr** and **LocalDataSetNum** commands to store information in the local property set for this user action.

14.122 LocalDataSetNum

LocalDataSetNum

This command sets a value in the user action's local property set to a given numeric value.

Each user action has its own local property set that lasts for as long as that instance of the action is running. A local property set consists of several named properties, and can be used as a convenient extension to the [**Data*n***] macro variables.

Syntax:

LocalDataSetNum(PropertyName, Value)

Parameters:

- **PropertyName**: This string value represents the name of an individual property that you want to set to the given value. If the given property does not exist, then it is automatically created.

- **Value**: The numerical value to store in the local property set.

  This value will remain in the local property set until the user action completes execution, or until it is overwritten with a call to **LocalDataSetNum** or **LocalDataSetStr**.

Example:

```
' Store the contents of [Data1] in property "TempValue".
LocalDatSetNum("TempValue", [Data1])
```

Notes:

- You can also use the **GlobalDataSetNum** command to set data in the global property set. The global property set allows for properties to be shared between running instances of user actions, and the data lasts until loses connection with .

- You can use the **LocalDataSetStr** command to store a string value in a property.

14.123 LocalDataSetStr

LocalDataSetStr

This command sets a value in the user action's local property set to a given string value.

Each user action has its own local property set that lasts for as long as that instance of the action is running. A local property set consists of several named properties, and can be used as a convenient extension to the [**Data*n***] macro variables.

Syntax:

LocalDataSetStr(PropertyName, Text)

Parameters:

- **PropertyName**: This string value represents the name of an individual property that you want to set to the given value. If the given property does not exist, it is automatically created.

- **Text**: The string value to store in the local property set.

  This value will remain in the local property set until the user action completes execution, or until it is overwritten with a call to **LocalDataSetNum** or **LocalDataSetStr**.

Example:

```
'Store the current account code in property "OldAccountCode".
LocalDatSetStr("OldAccountCode", [AccountCode])
```

Notes:

- You can also use the **GlobalDataSetStr** command to set data in the global property set. The global property set allows for properties to be shared between running instances of user actions, and the data lasts until loses connection with .

- You can use the **LocalDataSetNum** command to store a numerical value in a property.

14.124 MacroBtnRun

MacroBtnRun

This command executes one of the 12 button macros imported from CallViewer version 3. This command only exists for support of existing version 3 macros, because it only executes macros that were created in version 3.x.

Syntax:

MacroBtnRun(ButtonNumber)

Parameter:

ButtonNumber: This numerical value depicts the button number from CallViewer version 3 that should be executed. It is a value between 1 and 12.

Example:

```
' Run button macro 2.
MacroBtnRun(2)
```

14.125 MessageBox

MessageBox

This command displays a message to the user, and waits for the user to choose a button.

Syntax:

MessageBox(Message, ButtonType, Title)

Parameters:

- **Message**: The message to display to the user in the message box.

- **ButtonType**: This numerical value indicates the buttons that should be available in the message box. It can also be used to identify the type of message box to the user through an icon, as well identifying which button should be the default.

  The buttons to display are indicated by one of the following values:

Value	Button
0	Displays **OK** button only.
1	Displays **OK** and **Cancel** buttons.
2	Displays **Abort**, **Retry**, and **Ignore** buttons.
3	Displays **Yes**, **No**, and **Cancel** buttons.
4	Displays **Yes** and **No** buttons.
5	Displays **Retry** and **Cancel** buttons.

  If you want to display an icon on the message box, add one of the following values to the number:

Value	Icon
16	Displays a Stop icon.
32	Displays a Question Mark icon.
48	Displays an Exclamation Mark icon.
64	Displays an Information icon.

  If you want to assert the default button, add one of the following values:

Value	Default Button
0	First button is default.
256	Second button is default.
512	Third button is default.

  For example, specifying “4 + 32 + 256” would display Yes and No buttons on the message box (4), with a question mark icon

(32), and the second button would be the default button (256).

- **Title**: The text to display in the titlebar of the message box.

Example:

```
MessageBox("Click OK to continue.", 0+48, "")
```

Notes:

If you want to branch execution of the user action based on the button that the user clicks, you should use the **GotoIfMessageBox** command, which displays a message box to the user, and then branches execution based on the button clicked.

14.126 MessageBoxCustom

MessageBoxCustom

This command displays a message to the user, and waits for the user to choose a button. If the user has not chosen a button within a given time period, the message box is automatically closed.

Syntax:

MessageBox(Message, ButtonType, Title, Timeout)

Parameters:

- **Message**: The message to display to the user in the message box.

- **ButtonType**: This numerical value indicates the buttons that should be available in the message box. It can also be used to identify the type of message box to the user through an icon, as well identifying which button should be the default.

  The buttons to display are indicated by one of the following values:

Value	Button
0	Displays **OK** button only.
1	Displays **OK** and **Cancel** buttons.
2	Displays **Abort**, **Retry**, and **Ignore** buttons.
3	Displays **Yes**, **No**, and **Cancel** buttons
4	Displays **Yes** and **No** buttons.
5	Displays **Retry** and **Cancel** buttons.

  If you want to display an icon on the message box, add one of the following values to the number.

Value	Icon
16	Displays a Stop icon.
32	Displays a Question Mark icon.
48	Displays an Exclamation Mark icon.
64	Displays an Information icon.

  If you want to assert the default button, add one of the following values:

Value	Default Button
0	First button is default.
256	Second button is default.
512	Third button is default.

For example, specifying “4 + 32 + 256” would display Yes and No buttons on the message box (4), with a question mark icon (32), and the second button would be the default button (256).

- **Title**: The text to display in the titlebar of the message box.

- **Timeout**: This numerical value defines the number of milliseconds to wait before the message box is automatically closed. It can be a value between 0 and 60000 (60 seconds).

Example:

```
MessageBoxCustom(“Click OK to continue.”, 0+48, “”, 10000)
```

Notes:

If you want to branch execution of the user action based on the button that the user clicks, you should use the **GotoIfMessageBox** command, which displays a message box to the user, and then branches execution based on the button clicked.

14.127 MousePointer

MousePointer

This command changes the mouse cursor between an hourglass and the default pointer. It is a good idea to set the mouse cursor to display an hourglass if the user action is going to do something for a while such that the user will not be able to interact with the software.

Syntax:

MousePointer(DisplayHourglass)

Parameter:

DisplayHourglass: If this numerical value is set to 0, the mouse cursor is set to display the default cursor arrow.

If the value is set to 1, then the mouse cursor is set to display the hourglass cursor.

Example:

```
MousePointer(0)
```

Notes:

The mouse cursor is reset to the default cursor arrow when a user action completes execution.

14.128 MousePos

MousePos

This command sets the mouse cursor position on the screen to the given coordinates.

Syntax:

MousePos(x, y)

Parameters:

- **X**: This numerical value depicts the horizontal position on the screen of the mouse cursor, where 0 is the leftmost edge of the screen. The value is measured in pixels.

- **Y**: This numerical value depicts the vertical position on the screen of the mouse cursor, where 0 is the topmost edge of the screen. The value is measured in pixels.

Example:

```
MousePos(300, 100)
```

14.129 ODBCClose

ODBCClose

This command closes a previously opened ODBC connection. If the connection has not been opened, then an error is generated.

Syntax:

ODBCClose(Channel)

Parameter:

Channel: The channel number of the ODBC connection to close. This can be a value from 1 to 3.

Example:

```
' Close the first ODBC channel.
ODBCClose(1)
```

Notes:

By default, errors generated will stop execution of the macro. You can switch off this functionality with the SetErrorsFatal command. Having done so, the error is available using the [ErrorDesc] and [ErrorNum] macro variables.

14.130 ODBCGetField

ODBCGetField

This command retrieves a piece of data from the current record of the given ODBC connection. Having opened an ODBC connection with ODBCOpen, and then moved to the appropriate record with ODBCMove, you would use ODBCGetField to obtain the contents of a database field.

Syntax:

ODBCGetField(Channel, FieldName, DataVar)

Parameters:

- **Channel**: The channel number of the ODBC connection to read a field value from. This can be a value from 1 to 3.
- **FieldName**: The name of the field in the database whose value is to be retrieved. This name will depend on the database that you are trying to read information from.
- **DataVar**: This numerical value represents the number of the [Datan] macro variable that the result will be written to. This can be between 1 and 11.

Example:

```
' Move to the first record of channel 1
ODBCMove(1, 0, 0)
' Get the contents of field "LastContactDate" into [Data4]
ODBCGetField(1, "LastContactDate", 4)
```

Notes:

By default, errors generated will stop execution of the macro. You can switch off this functionality with the SetErrorsFatal command. Having done so, the error is available using the [ErrorDesc] and [ErrorNum] macro variables.

14.131 ODBCMove

ODBCMove

This command moves the current record pointer for the given ODBC connection to the required position.

Syntax:

ODBCMove(Channel, MoveType, MoveIndex)

Parameters:

- Channel: The channel number of the ODBC connection to move the current record pointer.. This can be a value from 1 to 3.
- MoveType: This is the type of move to perform, and can be one of the following values:

0	Move to the first record in the table
1	Move to the last record in the table
2	Move to the previous record in the table
3	Move to the next record in the table
4	Move to the given record in the “MoveIndex”

- MoveIndex: If the "MoveType" is 4, to move to a given record, then this field defines the record number to move to.

Example:

```
' Move to the last record in the results.
ODBCMove(1, 1)
```

Notes:

By default, errors generated will stop execution of the macro. You can switch off this functionality with the SetErrorsFatal command. Having done so, the error is available using the [ErrorDesc] and [ErrorNum] macro variables.

14.132 ODBCOpen

ODBCOpen

This command opens an ODBC connection on a given channel, and at the same time performs a query on the ODBC connection.

Syntax:

ODBCOpen(Channel, ConnectionString, QueryString)

Parameters:

- **Channel**: The channel number of the ODBC connection to open the connection on. This can be a value from 1 to 3. The channel should not have an active connection on it, otherwise an error will be generated.
- **ConnectionString**: This string is the connection string to be passed to ODBC. This defines the ODBC driver to use, other relevant information as to the location of the database, as well as the username and password. See the table below for some example connection strings.
- **QueryString**: This string defines the database query to depict what data to retrieve from the given database, e.g. "SELECT * FROM MyTable WHERE MyTable.ID=27;"

Connection Strings:

The connection string is very important as it defines the database type that is being connected to, and where it is. Some common example connection strings are as follows:

Microsoft Access (no workgroup file)	Driver={Microsoft Access Driver (*.mdb)}; Dbq=C:\MyDatabase.mdb;Uid=Admin;Pwd=;
Microsoft Access (with workgroup file)	Driver={Microsoft Access Driver (*.mdb)}; Dbq=C:\MyDatabase.mdb;SystemDB=C:\MyDatabase.mdw;
SQL Server	Driver={SQL Server}; Server=ServerName;Database=pubs;Uid=sa;Pwd=password;
Oracle	Driver={Microsoft ODBC for Oracle}; Server=ServerName;Uid=Username;Pwd=password;

You may need to contact the database administrator for information on what parameters are needed to successfully connect via ODBC to the database.

Example:

```
' Open a SQL Server database and query it for records that
' have a field that matches the Caller ID.
DataSetStr(1, "Driver={SQL Server);Server=MYDB;Database=pubs;Uid=sa;Pwd=;")
DataSetStr(2, "SELECT * FROM Customer WHERE Customer.Phone='" + [Digits] + "';")
ODBCOpen(1, [Data1], [Data2])
```

Notes:

- The ODBCOpen command blocks while it is accessing the database and performing the query. If the query is complex, or the database is busy, then this could take time. Usually, the more restricting the query, the fewer results returned, which can speed up execution of the command.

- By default, errors generated will stop execution of the macro. You can switch off this functionality with the SetErrorsFatal command. Having done so, the error is available using the [ErrorDesc] and [ErrorNum] macro variables. This can be useful when testing your macro, since it sometimes takes a couple of attempts to find the correct connection string for a database.

- Before calling ODBCMove to move to a given record, you should check that records have been returned at all using the GotoIfNoRecords command.

- You must close the connection with ODBCClose before attempting to open a new connection on the same channel, or before the action ends.

14.133 ODBCSetFieldNum

ODBCSetFieldNum

This command updates a given field for the current record in an open database with a numeric value.

Syntax:

ODBCSetFieldNum(Channel, FieldName, NewValue)

Parameters:

- **Channel**: The channel number of the ODBC connection to. This can be a value from 1 to 3.
- **FieldName**: The name of the field in the database table that you wish to set.
- **NewValue**: The numerical value that you wish to set the given field for the current record to.

Example:

```
' Increment the "Count" field in the table by 1
ODCBGetField(1, "Count", 0, 1)
ODBCSetFieldNum(1, "Count", [Data1] + 1)
```

Notes:

- By default, errors generated will stop execution of the macro. You can switch off this functionality with the SetErrorsFatal command. Having done so, the error is available using the [ErrorDesc] and [ErrorNum] macro variables.
- Some databases will be read-only, such that updating of fields will fail using this command.
- If you need to store a text-based value, use the ODBCSetFieldStr command instead.

14.134 ODBCSetFieldStr

ODBCSetFieldStr

This command updates a given field for the current record in an open database with a string value.

Syntax:

ODBCSetFieldStr(Channel, FieldName, NewValue)

Parameters:

- **Channel**: The channel number of the ODBC connection to. This can be a value from 1 to 3.
- **FieldName**: The name of the field in the database table that you want to set.
- **NewValue**: The text value for which you want to set the given field for the current record.

Example:

```
' Store the Caller ID in the "Last Call" field
ODBCSetFieldStr(1, "Last Call", [Digits])
```

Notes:

- By default, errors generated will stop execution of the macro. You can switch off this functionality with the SetErrorsFatal command. Having done so, the error is available using the [ErrorDesc] and [ErrorNum] macro variables.
- Some databases will be read-only, such that updating of fields will fail using this command.
- If you need to store a text-based value, use the ODBCSetFieldNum command instead.

14.135 PostMessage

PostMessage

This command puts a Windows message in a given window's message queue, and then returns without waiting for the corresponding window to process the message.

Syntax:

PostMessage(WindowTitle, Message, Param1, Param2)

Parameters:

- **WindowTitle**: The left part of the title of the application window to post a message to.

  The name that appears in the titlebar need not be fully specified. For instance, "Calc" would still find an open application with titlebar text "Calculator". The comparison is also not case sensitive; i.e., "Calculator" and "calculator" appear identical.

- **Message**: This numerical value identifies the Windows message to post to the given window. The actual values that can be used are beyond the scope of this help, but are available in the "Windows Platform SDK".

- **Param1**: This is additional message-dependent information, passed as the "wParam" parameter of the Windows message.

- **Param2**: This is additional message-dependent information, passed as the "lParam" parameter of the Windows message.

Example:

```
' Post WM_CLOSE to Notepad application.
PostMessage("Notepad", 16, 0, 0)
```

Notes:

- The use of this command is considered extremely advanced, and can lead to unexpected results. It is strongly recommended that this command not be used unless you are a qualified developer, with experience of Windows programming.

- The **SendMessage** command is similar, but waits for the message to be processed by the given window before continuing.

14.136 SendKeys

SendKeys

This command emulates sending a keystroke sequence to the currently active Windows-based application. Execution of the user action does not continue until the keystrokes have been processed.

Syntax:

SendKeys(Keystrokes)

Parameter:

Keystrokes: This string represents the keystrokes that are to be sent to the currently active application.

Most keystrokes are represented by each individual character in the string, for example the string “apple” would emulate pressing the “a” key, then “p,” “p,” “l,” and “e.”

You can emulate a key being pressed with Control, Shift, and/or Alt being pressed at the same time, with one of the following modifiers:

Character	Represents
Plus (+)	Shift key
Percent (%)	Alt key
Caret (^)	Control key

If you needed to emulate Control-Alt-X, you provide the text “^%x”. If you want to use the modifiers across several keys, enclose the keys in parentheses, e.g., “%(fa)” emulates Alt-F followed by Alt-A.

There are several keys that you cannot easily provide in the string, such as the Escape key, or the function keys. To emulate these keystrokes, use one of the following keywords in curly braces, e.g,. “{ENTER}”.

Keystroke	Keyword
Backspace	{BACKSPACE} or {BS} or {BKSP}
Break	{BREAK}
Caps Lock	{CAPSLOCK}
Clear	{CLEAR}
Del	{DELETE} or {DEL}
Down Arrow	{DOWN}
End	{END}
Enter	{ENTER} or ~ (tilde)
Esc	{ESCAPE} or {ESC}
Help	{HELP}
Home	{HOME}
Ins	{INSERT}

Left Arrow	{LEFT}
Num Lock	{NUMLOCK}
Page Down	{PGDN}
Page Up	{PGUP}
Right Arrow	{RIGHT}
Scroll Lock	{SCROLLLOCK}
Tab	{TAB}
Up Arrow	{UP}
F1	{F1}
F2	{F2}
F3	{F3}
F4	{F4}
F5	{F5}
F6	{F6}
F7	{F7}
F8	{F8}
F9	{F9}
F10	{F10}
F11	{F11}
F12	{F12}

If you need to emulate a key being pressed several times, then enclose the key or keyword in curly braces, followed by a space, and the number of times to repeat the key, e.g., “{ENTER 5}” would press the ENTER key five times.

If you need to type one of the special modifiers (%, ^, +, or ~), enclose them in curly braces too, e.g., “5% ” would press the 5 key followed by Alt+Space, whereas “5{%}” would type “5%”.

Example:

```
' Select all the text in the current control
' by pressing Control-End, then Control-Shift-Home.
SendKeys(“^{END}”)
SendKeys(“^+{HOME}”)
' Access the Edit menu (E), and choose cut (X)
SendKeys(“%EX”)
```

Notes:

- This command can only emulate keystrokes to native Windows applications. If you need to send keystrokes to a console window or MS-DOS application, use the **SendKeysEx** command instead.

- If you want to emulate keystrokes without waiting for the application to process the keystrokes, use the **SendKeysNoWait** command instead.

- Although square brackets (“[” and “]”) are not special characters in the keystrokes string, they can be interpreted as special in other applications, and so should be enclosed in curly braces too, e.g., rather than using “[Example]”, you would use “{[}Example{]}”.

14.137 SendKeysEx

SendKeysEx

This command emulates sending a keystroke sequence to the currently active application. Unlike the **SendKeys** command, **SendKeysEx** can be used to emulate keystrokes in both Windows applications, as well as MS-DOS and console application windows.

Syntax:

SendKeysEx(Keystrokes, PauseTime)

Parameters:

- **KeyStrokes**: This string represents the keystrokes that are to be sent to the currently active application. See the **SendKeys** command for information on the format of this string.

- **PauseTime**: This numerical value defines the number of milliseconds to pause between each keystroke. The value can be from 0 to 10000 (which is 10 seconds).

  It is recommended that a value other than 0 be used for this setting. This command is emulating keystrokes at the keyboard driver level, and while a human could not type 1000 keystrokes a second, **SendKeysEx** can! This can lead to the keyboard buffer being filled, and then keystrokes are lost. A value somewhere between 10 and 50 often works well, although experimentation with your chosen application is a good idea.

Example:

```
' Activate MS-DOS window.
AppActivateLike(“MS-DOS”)
YieldToOs
' Send DIR to MS-DOS window.
SendKeysEx(“DIR”, 50)
' Let MS-DOS screen repaint the actions
' caused by sending the keystrokes.
YieldToOs
' Send Enter key.
SendKeysEx(“ENTER”, 50)
```

Notes:

- Because this command emulates the keyboard at a driver level, it is a good idea to give the application receiving the keystrokes a chance to catch up. There are two ways of achieving this. First, place **YieldToOs** commands after keystrokes that might affect the screen updating, such as displaying menus or dialogs. Secondly, use several **SendKeysEx** commands with short keystroke strings, rather than one command with a really long keystroke string.

- There may be some console applications that still cannot receive emulated keystrokes from this command. In such an instance, use **ClipboardSetText** to copy your required text into the clipboard, and then use **SendKeysEx** to the console window to emulate using the system menu, and selecting Edit, then Paste.

14.138 SendKeysNoWait

SendKeysNoWait

This command emulates sending a keystroke sequence to the currently active Windows-based application. Unlike the **SendKeys** command, **SendKeysNoWait** continues execution on the next line of the script without waiting for the keystrokes to be processed.

Syntax:

SendKeysNoWait(Keystrokes)

Parameter:

KeyStrokes: This string represents the keystrokes that are to be sent to the currently active Windows-based application. See the **SendKey** command for information on the format of this string.

Example:

```
' Post a keystroke sequence to windows that would change
' the current MiCC Office Server if the Network Settings
' Window was open.
SendkeysNoWait("%S") ' Goto Server control.
SendkeysNoWait("CTISERVER2") ' Enter new MiCC Office Server name.
SendkeysNoWait("ENTER") ' Save the new settings.
' Open the Network Settings Window.
intSettingsNetwork
```

Notes:

If you want to emulate keystrokes and wait for the application to process the keystrokes, use the **SendKeys** command instead.

14.139 SendMessage

SendMessage

This command puts a Windows message in a given window's message queue, and then waits for the corresponding window to process the message.

Syntax:

SendMessage(WindowTitle, Message, Param1, Param2)

Parameters:

- **WindowTitle**: The left part of the title of the application window to send a message to.

  The name that appears in the titlebar need not be fully specified. For instance, "Calc" would still find an open application with titlebar text "Calculator". The comparison is also not case sensitive; i.e., "Calculator" and "calculator" appear identical.

- **Message**: This numerical value identifies the Windows message to send to the given window. The actual values that can be used are beyond the scope of this document, but are available in the "Windows Platform SDK."

- **Param1**: This is additional message-dependent information, passed as the "wParam" parameter of the Windows message.

- **Param2**: This is additional message-dependent information, passed as the "lParam" parameter of the Windows message.

Example:

```
' Immediately send WM_CLOSE to Notepad application.
SendMessage("Notepad", 16, 0, 0)
```

Notes:

- The use of this command is considered extremely advanced, and can lead to unexpected results. It is strongly recommended that this command not be used unless you are a qualified developer, with experience of Windows programming.

- The **PostMessage** command is similar, but does not wait for the message to be processed by the given window before continuing.

14.140 SetAccountCode

SetAccountCode

This command sets the account code on an external (trunk line) call at the given extension.

If there are no external trunk line calls at the given extension, an error occurs.

Syntax:

SetAccountCode(Extension, CallItem, AccountCode, UseCallControl)

Parameters:

- **Extension**: The extension device to set the account code at. If a blank string is specified, then the extension assigned to the running instance of will be used.

- **CallItem**: The index of the call to set the account code on. Calls in the list are identified as 1 for the first call in the call list, 2 for the second call etc. A value of 0 will instruct to automatically set the account code on the first external call at the given extension.

  In fact, if the call specified at the given extension is not an external call, the next call at the extension is used instead.

- **AccountCode**: The account code to set on the given call. If the “UseCallControl” parameter is 1, this parameter has a maximum length of 12 characters, otherwise it has a maximum length of 50 characters.

  The account code in this parameter will overwrite any existing account code active on this call.

- **UseCallControl**: If this numerical value is 0 then the account code bypasses the telephone system, and is processed only internally within . This allows for longer account codes, and support for the account code functionality across all telephone systems.

Example:

```
' Enter account code 999 at extension 200 via the
' telephone system.
SetAccountCode(“200”, 0, “999”, 1)
```

Notes:

You can generically find the device number of the extension associated with the current installation of t by using the [**LocalExtension**] macro variable.

14.141 SetACDAgentState

SetACDAgentState

This command changes that ACD agent state of a specified agent ID at the given extension.

Syntax:

SetACDAgentState(Extension, AgentID, AgentState, ACDGroup)

Parameters:

- **Extension**: The extension device to set the agent state at. If a blank string is specified, then the extension assigned to the running instance of will be used.
- **AgentID**: The ID that defines the agent whose state is to change.
- **AgentState**: This numerical value depicts the new agent state to place the given extension / agent in. It can be one of the following values:

Value	Description
0	
1	
2	Changes the specified agent's state to "Free."
3	Changes the specified agent's state to "Busy (Call)."
4	Changes the specified agent's state to "Busy (E-mail)."
5	Changes the specified agent's state to "Wrapup (Call)." This is performed in such a way that any telep system-based wrap-up timer is ignored, and as such the agent will remain in the wrap-up state indefin or at least until they enter the free state.
6	
7	Changes the specified agent's state to "Free (E-mail)."
8	Changes the specified agent's state to "Wrapup (E-mail)."

- **ACDGroup**:

Example:

```
' Log Agent 470 in at extension 200.
SetACDAgentState("200", "470", 1, "")
```

Notes:

- If you are changing the agent state at the extension assigned to , to a state other than log in or log out, you can use the [ACDAgentID] macro variable to obtain the Agent ID of the agent logged in to this extension right now.
- The agent must already be logged in when changing an agent's state to anything other than log in.

- You can generically find the device number of the extension associated with the current installation of by using the [**LocalExtension**] macro variable reference.

14.142 SetErrorsFatal

SetErrorsFatal

This command controls whether errors generated by the DDExxx, Filexxx, and ODBCxxx commands cause macro execution to be halted. By default, errors are fatal and so halt macro execution. Using this command to change this, causes the supported commands to carry on as if they had been successful, but store the resulting error and error code in macro variables so that they can be checked by the caller.

Syntax:

SetErrorsFatal(ErrorsFatal)

Parameters:

ErrorsFatal: If this numerical value is 1, then errors will be fatal. If an error occurs in the DDE, File, or ODBC commands, then macro execution will halt.

If this numerical value is 0, then errors will be ignored in the DDE, File, or ODBC commands. The macro will have access to the last error via the [ErrorDesc] and [ErrorNum] macro variables.

Example:

```
' Switch off errors before opening a file…
SetErrorsFatal(0)
FileOpen(1, "C:\MyFile.txt",1)
' Jump to "GotError" if [ErrorNum] isn't 0…
GotoIfNumVal("GotError", [ErrorNum], 0, 1)
```

Notes:

This command only affects the DDExxx, Filexxx, and ODBCxxx commands. All other commands will continue to be fatal if they generate an error.

14.143 SetForwardState

SetForwardState

This method changes the forward status of an extension. The method returns True if successful, or False otherwise.

Syntax:

SetForwardState Extension, ForwardType, Destination, Enable

Parameters:

- **Extension**: The extension device to set the forward state at. If a blank string is specified, then the extension assigned to the running instance of will be used.

- **ForwardType**: This numerical value defines the type of forwarding to set. It can be one of the following values:

Value	Description
0	Disables any forwarding or diverts at the given extension.
1	Enables / disables the forwarding of all calls to the forwarding device in “Destination.”
2	Enables / disables the forwarding of unanswered calls to the forwarding device in “Destination.”
3	Enables / disables the forwarding of calls when the device is busy to the forwarding device in “Destinatio
4	Enables / disables the forwarding of both unanswered calls and calls when the device is busy, to the forwarding device in “Destination.”

- **Destination**: This is the device that the call will be forwarded to, when enabling a forwarding type.

- **Enable**: If this value is False, the corresponding forwarding type is disabled.

  If the value is True, the corresponding forwarding type is enabled.

  This setting is ignored when setting forward type 0 on the Inter-Tel Axxess / Mitel 5000 Communications Platform System.

Example:

```
' Forward all calls at extension 200 to 213
axCallview.SetForwardState “200”, 1, “213”, True
' Disable forwarding of calls at extension 200
axCallview.SetForwardState “200”, 1, “”, False
' Forward calls at this extension to 299, if the device is busy
axCallview.SetForwardState “”, 3, “299”, True
```

14.144 SetIniSettingNum

SetIniSettingNum

This command is used to store or update a setting in the CVMACRO.INI file to a numerical value. The settings in this fle can be used to provide specifical configuration for user-defined macros.

Syntax:

SetIniSettingNum(SectionName, KeyName, NewValue)

Parameters:

- SectionName: This string value represents the name of the section in the INI file where the setting will be stored. Sections are denoted in INI files by wrapping them in square brackets, e.g. "[SectionName]".
- KeyName: This string value represents the name of the value in the INI file to set. The key to be set must fall within the section denoted by the "SectionName" parameter.
- NewValue: This numeric value represents the data to be written to the setting in the file.

Example:

```
' Store a number in the "SearchAllPhoneFields" option of the
' [Options] section.
SetIniSettingNum("Options", "SearchAllPhoneFields", 1)
```

Notes:

You can retrieve settings from the CVMACRO.INI file using the GetIniSetting command.

14.145 SetIniSettingStr

SetIniSettingStr

This command is used to store or update a setting in the CVMACRO.INI file to a text-based value. The settings in this fle can be used to provide specifical configuration for user-defined macros.

Syntax:

SetIniSettingStr(SectionName, KeyName, NewValue)

Parameters:

- **SectionName**: This string value represents the name of the section in the INI file where the setting will be stored. Sections are denoted in INI files by wrapping them in square brackets, e.g. "[SectionName]".
- **KeyName**: This string value represents the name of the value in the INI file to set. The key to be set must fall within the section denoted by the "SectionName" parameter.
- **NewValue**: This text-based value represents the data to be written to the setting in the file.

Example:

```
' Store the last CLI in the "CLI" key of the
' "Last Call Info" section.
SetIniSettingStr("Last Call Info", "CLI", [Digits])
```

Notes:

You can retrieve settings from the CVMACRO.INI file using the GetIniSetting command.

14.146 SetKeyState

SetKeyState

This command is used to set the key state of a given key on the keyboard. This can be used to toggle Num Lock and Caps Lock on and off

Syntax:

SetKeyState(Key, State)

Parameters:

- **Key**: This is a numerical value that depicts which key is going to be toggled. Potential values are as follows:

Value	Key
20	Caps Lock
144	Num Lock
145	Scroll Lock

- **State**: If this numerical value is 0, the given key is toggled off, e.g., caps lock is turned off.

  If the value is 1, the given key is toggled, so if it is on it is switched off, and vice versa.

Example:

```
' Set the toggle state of the Num Lock key on.
SetKeyState(144, 1)
```

Notes:

Although this command can be used to simulate key presses, you are recommended to use **SendKeys** or **SendKeysEx** to simulate key presses.

14.147 SetStatusLine

SetStatusLine

This command sets the text being displayed on the status line of CallViewer .

Syntax:

SetStatusLine(StatusText)

Parameter:

StatusText: The text to display in the status bar.

Example:

```
SetStatusLine(“Running user action...”)
```

Notes:

Not every “Look and Feel” has a status bar. An error is not thrown if calling this command with a window that has no status bar.

14.148 SetTrunkCallParam

SetTrunkCallParam

This command updates a piece of information against the current call. This can be useful for using a call's Caller ID to identify an unrecognized contact, and then updating the "Field2" property against the call, so that the contact becomes recognized .

Syntax:

SetTrunkCallParam(Extension, CallIndex, Parameter, NewValue)

Parameters:

- **Extension**: The extension device to set the forward state at. If a blank string is specified, then the extension assigned to the running instance of Callviewer will be used.
- **CallIndex**: The call index of the call you wish to update. This can be 0, which will cause Client to pick the first available call and update that.
- **Parameter**: This value depicts which field you want to update against the given call, as follows:

0	Telephone Number
1	Field 2, usually Company Name
2	Field 3 (user-defined)
3	Field 4 (user-defined)
4	Field 5 (user-defined)
5	Field 6 (user-defined)

- **NewValue**: This string value depicts the new value to store against this item of the call.

Example:

```
' Update Field2 of the current call with the results of the
' DDE query on DDE channel 1…
SetTrunkCallParam("", 0, 1, [DDE1])
```

Notes:

If an unrecognized call is updated such that the "Field2" parameter is non-blank, then the Server will treat the call as having been recognized , which will improve the historical and real-time statistics related to calls recognized . However, when the same caller rings in again, they will not be identified, and the macro command would have to be used again in a similar fashion.

14.149 SetVolume

SetVolume

This command changes the volume of the given extension.

Syntax:

SetVolume(Extension, VolumeType, VolumeLevel, Save)

Parameters:

- **Extension**: The extension device to set the volume at. If a blank string is specified, then the extension assigned to the running instance of CallViewer will be used.

- **VolumeType**: This numerical value defines the type of volume that is to be changed.

Value	Description
0	Context Specific – this option changes the volume based on the current state of the extension, e.g., if extension is alerting, the “alerting” volume is altered.
1	Alerting volume.
2	Off hook tone volume.
3	Internal call volume.
4	External call volume.

  Currently, only the “context specific” option is supported by the telephone systems.

- **VolumeLevel**: This numerical value depicts the new volume level as follows:

  A value of 1 increases the volume level by 1, while a value of 2 decreases the volume level by 1. You can also use a value of 0 that does not change the volume level, but which can be used with the “Save” parameter to save the current volume level.

- **Save**: This numerical value is 1 if the new volume level is to be saved for the given volume type. If it is 0, the volume level is changed, but not persisted. After a new call starts, the volume will be at its previous level.

Example:

```
' Increase the volume at extension 200
' by one level (on the Inter-Tel Axxess).
SetVolume(“200”, 0, 1, 1)
```

Notes:

T his command requires OAI Protocol V05.10 or later, or MiTAI.

14.150 Shell

Shell

This command launches the given application, specified using its filename. If the file cannot be found, or a problem occurs launching the file, an error is generated.

Syntax:

Shell(Filename)

Parameter:

Filename: This string value is the fully pathed filename of the application to launch, e.g., C:\WINDOWS\NOTEPAD.EXE.

Example:

```
' Run Microsoft Access.
Shell("C:\MSOFFICE\ACCESS\MSACCESS.EXE")
```

Notes:

- The **ShellEx** command provides more functionality than this command, allowing you to pass command line parameters as well.
- Using the **YieldToOs** command after the application launches will give it time to itself before subsequent script lines are processed.

14.151 ShellEx

ShellEx

This command launches an application with specific command line options. If the application file cannot be found, or a problem occurs launching the file, an error is generated.

Syntax:

ShellEx(Filename, CommandLine)

Parameters:

- **Filename**: This string value is the fully pathed filename of the application to launch, e.g., C:\WINDOWS\NOTEPAD.EXE.

- **CommandLine**: This string is the command line parameters to pass to the application. The contents of this string will depend on the application being launched.

Example:

```
' Run Microsoft Access and load the customer database.
ShellEx("C:\MSOFFICE\ACCESS\MSACCESS.EXE", "C:\DBASE\CUST.MDB")
```

Notes:

Using the **YieldToOs** command after the application launches will give it time to itself before subsequent script lines are processed.

14.152 Wait

Wait

This command waits for a given number of milliseconds to elapse before execution of the user action continues

Syntax:

Wait(Duration)

Parameter:

Duration: This numeric value depicts the time in milliseconds that the user action should wait for.

Example:

```
' Pause for 1 second.
Wait(1000)
```

14.153 WaitAppTitle

WaitAppTitle

This command pauses execution of the user action until a given application window changes its titlebar text to a given string. If the specified window cannot be found, an error will occur.

Syntax:

WaitAppTitle(WindowTitle, NewWindowTitle, IgnoreErrorFlag)

Parameters:

- **WindowTitle**: The left part of the title of the application window to pause execution for until the title changes.

  The name that appears in the titlebar need not be fully specified. For instance, “Calc” would still activate an open application with titlebar text “Calculator”. The comparison is also not case sensitive; i.e., “Calculator” and “calculator” appear identical.

- **NewWindowTitle**: The left part of the new title of the application window. When the titlebar text’s leftmost part matches this string, execution of the user action will continue.

  The name that appears in the titlebar need not be fully specified. For instance, “Calc” would still activate an open application with titlebar text “Calculator.” The comparison is also not case sensitive; i.e., “Calculator” and “calculator” appear identical.

- **IgnoreErrorFlag**: If this numerical value is 0, any errors that occur will cause execution of the user action to stop.

  If the value is 1, any errors will be ignored, and execution will continue on the next line of the script.

Example:

```
AppActivateLike(“MS-DOS”) ' Activate MS-DOS window.
YieldToOS
' Place 5 key Escape sequence into clipboard.
ClipboardSetText({27 5})
SendKeys(“% EP”) ' “Paste” into application.
' Pause until processed.
WaitAppTitle(“Paste MS-DOS”, “MS-DOS”, 1)
```

Notes:

- This command can be useful when integrating with MS-DOS or Console-based applications and the clipboard paste method of inserting text into the window is being . When keystrokes are sent this way to an MS-DOS or Console window, they are not processed immediately so keystrokes sent later in a user action are completely ignored. To combat this, execution needs to be paused until the MS-DOS or Console window has finished processing the keystrokes.

- While the MS-DOS/Console window processes the keystrokes, it displays “Paste - ” on the application window's titlebar in front of the normal titlebar text. You can use the **WaitAppTitle** command to pause execution until the titlebar text changes back to normal again.

- You should always try to use the **SendkeysEx** command to send MS-DOS or Console based applications keystrokes. You should only use the clipboard paste method if the **SendkeysEx** command cannot be used.

- This command can hang the Windows environment indefinitely should the titlebar text of a window not change. A better command to use instead is **WaitAppTitleTimeOut**. This allows the command to timeout after a given period, should the titlebar text not change.

14.154 WaitAppTitleTimeOut

WaitAppTitleTimeOut

This command pauses execution of the user action until a given application window changes its titlebar text to a given string. This command differs from **WaitAppTitle** by allowing for a user-definable timeout to occur, should the titlebar text not change.

If the specified window cannot be found, an error will occur.

Syntax:

WaitAppTitleTimeOut(WindowTitle, NewWindowTitle, IgnoreErrorFlag, Timeout, TimeoutNotify)

Parameters:

- **WindowTitle**: The left part of the title of the application window to pause execution for until the title changes.

  The name that appears in the titlebar need not be fully specified. For instance, “Calc” would still activate an open application with titlebar text “Calculator”. The comparison is also not case sensitive; i.e., “Calculator” and “calculator” appear identical.

- **NewWindowTitle**: The left part of the new title of the application window. When the titlebar text’s leftmost part matches this string, then execution of the user action will continue.

  The name that appears in the titlebar need not be fully specified. For instance, “Calc” would still activate an open application with titlebar text “Calculator”. The comparison is also not case sensitive; i.e., “Calculator” and “calculator” appear identical.

- **IgnoreErrorFlag**: If this numerical value is 0, any errors that occur will cause execution of the user action to stop.

  If the value is 1, any errors will be ignored, and execution will continue on the next line of the script.

- **Timeout**: The duration, in milliseconds, to wait for the titlebar text to change. The value that should be used here will depend on the reason why you are waiting for the titlebar to change, although extremely short or extremely long values are not advisable.

- **TimeoutNotify**: If this numerical value is 0 when a timeout occurs, execution will continue on the next line of the script, as if the titlebar text had changed.

  If the value is 1, a timeout will generate an error and stop execution of the user action.

Example:

```
AppActivateLike(“MS-DOS”) ' Activate MS-DOS window.
YieldToOS
' Place 5 key Escape sequence into clipboard.
ClipboardSetText({27 5})
SendKeys(“% EP”) ' “Paste” into application.
' Wait up to 1 second for keystrokes to get processed, but
' continue anyway if 1 second elapse period is reached.
WaitAppTitleTimeOut(“Paste MS-DOS”, “MS-DOS”, 1, 1000, 0)
```

Notes:

- This command can be useful when integrating with MS-DOS or Console based applications and the clipboard paste method of inserting text into the window is being . When keystrokes are sent this way to an MS-DOS or Console window, they are not processed immediately so keystrokes sent later in a user action are completely ignored. To combat this, execution needs to be paused until the MS-DOS or Console window has finished processing the keystrokes.

- While the MS-DOS/Console window processes the keystrokes, it displays “Paste - ” on the application window's titlebar in front of the normal titlebar text. You can use the **WaitAppTitleTimeout** command to pause execution until the titlebar text changes back to normal again.

- You should always try to use the **SendKeysEx** command to send MS-DOS or Console based applications keystrokes. You should only use the clipboard paste method if the **SendKeysEx** command cannot be used.

14.155 YieldToOs

YieldToOs

This command causes to yield the user action execution so that Windows can process events and finish multi-tasking operations.

This is usually required when the user action has just performed some form of integration with another application, e.g., selecting a menu item, or opening a window. Such activities will require some processing by the other application, and if the user action just continued straight away with the next line of the script, then the other application may not be ready to receive further commands, and so the user action would fail. By calling **YieldToOs**, the user action allows the other application to process any outstanding requests it has, such as completing the opening of a window, repainting the screen, or finishing off a menu item selection.

You would normally call **YieldToOs** after any **SendKeys** or **SendKeysEx** command, as well as any of the **App*XXX*** commands that open or move windows.

Syntax:

YieldToOS

Example:

```
' Launch Notepad
Shell("C:\WINDOWS\NOTEPAD.EXE")
' Wait for the application to open
YieldToOs
' Now send keystrokes to application...
```

Notes:

- Some operations may take a long time to complete, e.g., a search in a database. **YieldToOs** returns as soon as the application finishes processing events, which will normally be before a search or similar activity completes. In such scenarios you may have to use Wait to pause execution, however that is very indeterminate.

- You may find that the application provides some form of notification when such a process as a search completes, e.g., a dialog opens, or a titlebar changes, or a DDE variable is set. It is better to use something determinate, such as one of these examples, to judge when such a long process has completed. **YieldToOs** is more for ensuring that Windows is ready to accept user input again.

- When first performing an integration, you are better off using **YieldToOs** extensively, and keeping keystrokes sent via **SendKeys** or **SendKeysEx** to short sequences of keystrokes. The overhead on **YieldToOs** is quite small, so there is little or no performance overhead from calling it repeatedly, and it will ensure that your user action is more likely to work on all systems, rather than just working on your test system, and failing on other user's computers.

14.156 Macro Variables

This section details the macro variables that are available in the Callview Macro language. Remember that variables are only treated as variables when enclosed in square brackets, e.g., [AccountCode].

14.156.1 AreaPrefix

AreaPrefix

This is a string representing the local area telephone number prefix. It is the value entered into the “Local Area Prefix” setting within MiCC Office Server .

This string is blank if CallViewer is not connected to the MiCC Office Server .

14.156.2 AccountCode

AccountCode

This is a string value that depicts the last account code entered in against the current call in the call list.

For an action that is executed because a rule fired, the current call is the call in the call list that caused the rule to fire. If a rule did not execute the action, the current call is the selected call in the call list.

You can select a particular call in the call list using the **CallSelect** command.

14.156.3 ACDAgentID

ACDAgentID

This is a string value that depicts the ACD agent ID that is currently logged in to the extension device that CallViewer is associated with.

If the extension is not logged in with an agent ID, this variable returns an empty string.

14.156.4 ACDLoginCnt

ACDLoginCnt

This is a variable depicting the number of times that the extension associated with CallViewer has logged into a “non-agent ID” type hunt group.

14.156.5 ACDLoginCntAgID

ACDLoginCntAgID

This is a variable depicting the number of times that an ACD agent has logged into an “agent ID” type hunt group at the extension device that CallViewer is associated with in the current session.

14.156.6 ACDStatus

ACDStatus

This is a variable depicting the ACD agent status of the extension device that CallViewer is associated with.
The value can be one of the following:

Value	Description
0	Logged Out
1	Logged In
2	Free
3	Busy (Call)
4	Busy (E-mail)
5	Wrapup (Call)
6	Busy N/A (DND)
7	Wrapup (E-mail)
8	Free (E-mail)

14.156.7 CallAns

CallAns

This is a variable that equals 1 if the current call is answered, or 0 if unanswered.

For an action that is executed because a rule fired, the current call is the call in the call list that caused the rule to fire. If a rule did not execute the action, the current call is the selected call in the call list.

You can select a particular call in the call list using the **CallSelect** command.

14.156.8 CallAnsTime

CallAnsTime

This is a string value that depicts the answer date/time for the current call in the call list. The call answer date/time is in long format as defined in the International/Regional Settings section of the Windows Control Panel. If the current call is not answered, this variable returns an empty string.

For an action that is executed because a rule fired, the current call is the call in the call list that caused the rule to fire. If a rule did not execute the action, the current call is the selected call in the call list.

You can select a particular call in the call list using the **CallSelect** command.

14.156.9 CallCLI

CallCLI

This is a variable that equals 1 if was received for the current call, or 0 if no was received.

For an action that is executed because a rule fired, the current call is the call in the call list that caused the rule to fire. If a rule did not execute the action, the current call is the selected call in the call list.

You can select a particular call in the call list using the **CallSelect** command.

14.156.10 CallContact

CallContact

This is a variable that equals 1 if the received or digits for the current call was identified by the Telephone Number Import, or 0 if the contact was not identified. Calls that were not received with will always return 0 for this variable.

For an action that is executed because a rule fired, the current call is the call in the call list that caused the rule to fire. If a rule did not execute the action, the current call is the selected call in the call list.

You can select a particular call in the call list using the **CallSelect** command.

14.156.11 CallCtrl

CallCtrl

This is a variable that equals 1 if CallViewer ’s call control functionality is currently enabled, or 0 if it is disabled.

Call control can be enabled or disabled from the Enable Call Control option on the Call Control tab of the Options dialog. If the setting is disabled, no call control capability is available in CallViewer , either from the CallViewer Macro Language or from the CallViewer user interface.

14.156.12 CallHeld

CallHeld

This is a variable that equals 1 if the current call is on hold, of 0 if the call is not held.

For an action that is executed because a rule fired, the current call is the call in the call list that caused the rule to fire. If a rule did not execute the action, the current call is the selected call in the call list.

You can select a particular call in the call list using the **CallSelect** command.

14.156.13 CallInt

CallInt

This is a variable that equals 1 if the current call is an internal one or 0 if the call is external.

For an action that is executed because a rule fired, the current call is the call in the call list that caused the rule to fire. If a rule did not execute the action, the current call is the selected call in the call list.

You can select a particular call in the call list using the **CallSelect** command.

14.156.14 CallMediaType

CallMediaType

This is a string value that represents the type of media for the contact that caused the rule to execute this user action. It can be one of the following values:

Value	Description
CALL	The contact is a telephone call. A particular call event caused the rule to automatically execute this actio Alternatively, a call was selected in the CallViewer call list if the action was not executed by a rule.
EMAIL	The contact is a routed e-mail message. A particular e-mail event caused the rule to automatically execu this action.

14.156.15 CallOut

CallOut

This is a variable that equals 1 if the current call is an outbound call, or 0 if the call is inbound.

For an action that is executed because a rule fired, the current call is the call in the call list that caused the rule to fire. If a rule did not execute the action, the current call is the selected call in the call list.

You can select a particular call in the call list using the **CallSelect** command.

Note: The direction of the call is determined by how the call started on the current extension. For example, an external call may have been to an external party, but subsequently transferred to another extension; in such an instance the call is considered inbound for the receiving extension, since it received a call, rather than made a call.

14.156.16 CallRingTime

CallRingTime

This is a variable that depicts the ring time in seconds for the current call in the call list. If the current call is not answered, this variable returns the number of seconds that the call has been waiting for.

For an action that is executed because a rule fired, the current call is the call in the call list that caused the rule to fire. If a rule did not execute the action, the current call is the selected call in the call list.

You can select a particular call in the call list using the **CallSelect** command.

14.156.17 Calls

Calls

This is the number of active calls in the call list at this moment in time.

14.156.18 CallSelected

CallSelected

This is a variable that represents the index of the currently selected call in the call list. The first call in the list is “1,” the next “2,” and so on. If no call is selected in the call list, the value is “0.”

You can select a particular call in the call list using the **CallSelect** command.

14.156.19 CallSerialNo

CallSerialNo

This is the unique serial number of the current call in the call list. MiCC Office Server generates the serial number internally.

For an action that is executed because a rule fired, the current call is the call in the call list that caused the rule to fire. If a rule did not execute the action, the current call is the selected call in the call list.

You can use call serial number to tag records in your database, and subsequently map your database records to call log information in the MiCC Office Server databases. The serial number is written to the “TTSerialNo” field in the MiCC Office Server databases.

14.156.20 CallSource

CallSource

This variable is similar to the [**CallSelected**] variable, in that it returns the index of the current call in the call list. However, if the action was executed because a rule fired, this variable returns the index of the call that caused the rule to fire, regardless of the selected call in the call list.

Its main use is for actions that are executed because a rule fired, where it returns the index of the call.

14.156.21 CallStartTime

CallStartTime

This is a string value that depicts the start date/time for the current call in the call list. The date/time is in the long date format as defined in the International / Regional section of the Windows Control Panel.

For an action that is executed because a rule fired, the current call is the call in the call list that caused the rule to fire. If a rule did not execute the action, the current call is the selected call in the call list.

You can select a particular call in the call list using the **CallSelect** command.

14.156.22 CallWasOnHold

CallWasOnHold

This is a variable that equals 1 if the current call was on hold when last received an update about the call from . If the call was not on hold at the time of the last update, this variable equals 0.

For an action that is executed because a rule fired, the current call is the call in the call list that caused the rule to fire. If a rule did not execute the action, the current call is the selected call in the call list.

You can select a particular call in the call list using the **CallSelect** command.

14.156.23 CanCallAnswer

CanCallAnswer

This is a variable that equals 1 if the Answer call control feature is available at this moment, or equals 0 if the Answer feature is not currently available.

Typically, answering is available if the following conditions are met:

- There is a call alerting the extension associated with CallViewer .
- Call control is enabled in CallViewer ’s options.
- The telephone driver supports answering calls via call control.
- The license obtained by CallViewer allows calls to be answered.

14.156.24 CanCallConf

CanCallConf

This is a variable that equals 1 if the Conference call control feature is available at this moment, or equals 0 if the feature is not currently available.

Typically, conference is available if the following conditions are met:

- There are answered or held calls at the extension associated with CallViewer .
- Call control is enabled in CallViewer ’s options.
- The telephone driver supports conferencing calls via call control.
- The license obtained by CallViewer allows calls to be conferenced.

14.156.25 CanCallDial

CanCallDial

This is a variable that equals 1 if the Dial call control feature is available at this moment, or equals 0 if the feature is not currently available. The Dial feature enables the user to make calls from their extension.

Typically, dialing is available if the following conditions are met:

- The extension associated with CallViewer is idle, or has an exclusively held call present.
- Call control is enabled in CallViewer ’s options.
- The telephone driver supports making calls via call control.
- The license obtained by CallViewer allows calls to be dialed .

14.156.26 CanCallDialDig

CanCallDialDig

This is a variable that equals 1 if the Dial Digits call control feature is available at this moment, or equals 0 if the feature is not currently available.

Typically, dialing of digits is available if the following conditions are met:

- There is an answered call at the extension associated with CallViewer .
- Call control is enabled in CallViewer ’s options.
- The telephone driver supports dialing digits via call control.
- The license obtained by CallViewer allows digits to be dialed .

14.156.27 CanCallDrop

CanCallDrop

This is a variable that equals 1 if the Drop call control feature is available at this moment, or equals 0 if the feature is not currently available.

Typically, dropping calls is available if the following conditions are met:

- There is an outbound or answered call at the extension associated with CallViewer .
- Call control is enabled in CallViewer ’s options.
- The telephone driver supports dropping calls via call control.
- The license obtained by CallViewer allows calls to be dropped.

14.156.28 CanCallDropAll

CanCallDropAll

This is a variable that equals 1 if the Drop All (Release) call control feature is available at this moment, or equals 0 if the feature is not currently available.

Typically, dropping all calls is available if the following conditions are met:

- Call control is enabled in CallViewer ’s options.
- The telephone driver supports the handset being reset via call control.
- The license obtained by CallViewer allows the handset to be reset via call control.

14.156.29 CanCallHoldEx

CanCallHoldEx

This is a variable that equals 1 if the Exclusive Hold call control feature is available at this moment, or equals 0 if the feature is not currently available.

Typically, exclusively holding calls is available if the following conditions are met:

- There is an external outbound, or answered call at the extension associated with CallViewer .
- Call control is enabled in CallViewer ’s options.
- The telephone driver supports exclusively holding calls via call control.
- The license obtained by CallViewer allows calls to be held exclusively.

14.156.30 CanCallHoldSys

CanCallHoldSys

This is a variable that equals 1 if the System Hold (park) call control feature is available at this moment, or equals 0 if the feature is not currently available.

Typically, system-holding calls is available if the following conditions are met:

- There is an external outbound, or answered call at the extension associated with CallViewer .
- Call control is enabled in CallViewer ’s options.
- The telephone driver supports system holding calls via call control.
- The license obtained by CallViewer allows calls to be system held.

14.156.31 CanCallRetrieve

CanCallRetrieve

This is a variable that equals 1 if the Retrieve From Hold call control feature is available at this moment, or equals 0 if the feature is not currently available.

Typically, retrieving calls is available if the following conditions are met:

- There is an exclusively held call at the extension associated with CallViewer .
- Call control is enabled in CallViewer ’s options.
- The telephone driver supports retrieving calls via call control.
- The license obtained by CallViewer allows calls to be retrieved.

14.156.32 CanCallTrans

CanCallTrans

This is a variable that equals 1 if the Enquiry Transfer call control feature is available at this moment, or equals 0 if the feature is not currently available.

Typically, enquiry transfer is available if the following conditions are met:

- There is an external outbound, or answered call at the extension associated with CallViewer .
- Call control is enabled in CallViewer ’s options.
- The telephone driver supports enquiry transfer of calls via call control.
- The license obtained by CallViewer allows for enquiry transfer of calls.

14.156.33 CanCallTransComp

CanCallTransComp

This is a variable that equals 1 if the Complete Transfer call control feature is available at this moment, or equals 0 if the feature is not currently available.

Typically, complete transfer is available if the following conditions are met:

- There is a previously set-up consultation call at the extension associated with CallViewer , which is external outbound, or answered.
- Call control is enabled in CallViewer ’s options.
- The telephone driver supports complete transfer via call control.
- The license obtained by CallViewer allows for transfer completion of calls.

14.156.34 CanCallTransRedir

CanCallTransRedir

This is a variable that equals 1 if the Transfer/Redirect call control feature is available at this moment, or equals 0 if the feature is not currently available.

Typically,transfer/redirection of calls is available if the following conditions are met:

- There is an external or answered call at the extension associated with CallViewer .
- Call control is enabled in CallViewer ’s options.
- The telephone driver supports transferring / redirecting calls via call control.
- The license obtained by CallViewer allows calls to be transferred / redirected.

14.156.35 ConfPartyLimit

ConfPartyLimit

This is a variable that equals the maximum number of parties that are permitted in a call conference by the telephone system that Contact Center Server is connected to.

The maximum number of parties specified includes the extension that instigated the conference, i.e., the extension associated with CallViewer , therefore the maximum number of calls that can be included in the conference is one less than the number specified.

14.156.36 ClientActive

ClientActive

This is a variable that equals 1 if CallViewer is connected to the Contact Center Server , or 0 if it is not connected.

14.156.37 ClientName

ClientName

This variable provides a string value consisting of the text “EXT-” followed by the extension that CallViewer is associated with.

It is provided for backward compatibility with earlier versions of CallViewer , which used this value as the network name for the running instance of CallViewer .

14.156.38 ClientNameNum

ClientNameNum

This variable provides a string value depicting the extension that CallViewer is associated with. The variable is provided for backward compatibility to previous versions of CallViewer .

14.156.39 Clipboard

Clipboard

This variable provides a string value representing the text in the Windows clipboard. If there is no text in the clipboard, or the object in the clipboard cannot be rendered as text, then a zero length string is returned.

You can use the **ClipboardSetText** command to set the contents of the clipboard.

14.156.40 Col1

Col1

This variable equals the line or extension number for the current call in the call list.

For an action that is executed because a rule fired, the current call is the call in the call list that caused the rule to fire. If a rule did not execute the action, then the current call is the selected call in the call list.

You can select a particular call in the call list using the **CallSelect** command.

14.156.41 Col2

Col2

This variable contains call information that changes dependent on the type of the current call in . For external calls, this will display the DNIS string associated with the that the call came in on; for external non- calls it will display the description of the trunk line that the call is active on, otherwise for internal calls it will contain “[Internal].”

For an action that is executed because a rule fired, the current call is the call in the call list that caused the rule to fire. If a rule did not execute the action, the current call is the selected call in the call list.

You can select a particular call in the call list using the **CallSelect** command.

14.156.42 Col3

Col3

This variable equals information on Field 2 from the Telephone Import for the current call. Field 2 usually contains the company or caller name of the identified contact. If the caller could not be identified, this field will contain “[New Contact].”

For an action that is executed because a rule fired, the current call is the call in the call list that caused the rule to fire. If a rule did not execute the action, the current call is the selected call in the call list.

You can select a particular call in the call list using the **CallSelect** command.

14.156.43 Col4

Col4

This variable equals information on Field 3 from the Telephone Import for the current call. The contents of this field are dependent on the configuration of the import file on the . If the caller could not be identified, this field will contain an empty string.

For an action that is executed because a rule fired, the current call is the call in the call list that caused the rule to fire. If a rule did not execute the action, then the current call is the selected call in the call list.

You can select a particular call in the call list using the **CallSelect** command.

14.156.44 Col5

Col5

This variable equals information on Field 4 from the Telephone Import for the current call. The contents of this field are dependent on the configuration of the import file on the . If the caller could not be identified, this field will contain an empty string.

For an action that is executed because a rule fired, the current call is the call in the call list that caused the rule to fire. If a rule did not execute the action, then the current call is the selected call in the call list.

You can select a particular call in the call list using the **CallSelect** command.

14.156.45 Col6

Col6

This variable equals information on Field 5 from the Telephone Import for the current call. The contents of this field are dependent on the configuration of the import file on the . If the caller could not be identified, this field will contain an empty string.

For an action that is executed because a rule fired, the current call is the call in the call list that caused the rule to fire. If a rule did not execute the action, then the current call is the selected call in the call list.

You can select a particular call in the call list using the **CallSelect** command.

14.156.46 Col7

Col7

This variable equals information on Field 6 from the Telephone Import for the current call. The contents of this field are dependent on the configuration of the import file on the , but it is usually recommended that this contains the primary key for the related record in the company database. If the caller could not be identified, this field will contain an empty string.

For an action that is executed because a rule fired, the current call is the call in the call list that caused the rule to fire. If a rule did not execute the action, the current call is the selected call in the call list.

You can select a particular call in the call list using the **CallSelect** command.

14.156.47 CTIServerName

CTIServerName

This variable equals the network name of the Contact Center Server that CallViewer is connected to.

14.156.48 Data1

Data1

This variable stores temporary data assigned using such commands as **DataSetStr** or **DataSetNum**. There are 11 such variables, which can each store data independently. The contents of this variable are reset when execution of a user action ends.

When a user action is executed by a rule firing, this variable will initially contain information specific to that rule. If not executed by a rule, this does not apply. The table below indicates the contents of this variable based on the rule that caused the action to fire.

Rule Firing Event	[Data1] Contents
Call	The trunk line number that the call was active on.
ACD Agent Status Changed	A number between 0 and 8 depicting the status that the ACD agent changed to, as follows: 0 – Logged Out 1 – Logged In 2 – Free 3 – Busy (Call) 4 – Busy (E-mail) 5 – Wrapup (Call) 6 – Busy N/A (DND) 7 – Wrapup (E-mail) 8 – Free (E-mail)
ACD Agent Help	
Account Code Entered	The account code entered
Digits To Voice Mail	
Forward / Divert Status Changed	A number between 0 and 4 representing the forward / divert state being changed to, as follows: 0 – None 1 – Immediate 2 – No Answer 3 – On Busy 4 – No Answer / On Busy
Do Not Disturb Status Changed	

Note: If you subsequently call a command that sets the contents of this variable, the initial information pertaining to the rule will be lost.

14.156.49 Data10

Data10

This variable stores temporary data assigned using such commands as **DataSetStr** or **DataSetNum**. There are 11 such variables, which can each store data independently. The contents of this variable are reset when execution of a user action ends.

When a user action is executed by a rule firing, this variable will initially contain information specific to that rule. If not executed by a rule, this does not apply. The table below indicates the contents of this variable based on the rule that caused the action to fire.

Rule Firing Event	[Data10] Contents
Call	Contains field 6 from the matched record in the Telephone Import database.
ACD Agent Status Changed	Empty.
ACD Agent Help	Empty.
Account Code Entered	Empty.
Digits To Voice Mail	Empty.
Forward / Divert Status Changed	Empty.
Do Not Disturb Status Changed	Empty.

Note: If you subsequently call a command that sets the contents of this variable, the initial information pertaining to the rule will be lost.

14.156.50 Data11

Data11

This variable stores temporary data assigned using such commands as **DataSetStr** or **DataSetNum**. There are 11 such variables, which can each store data independently. The contents of this variable are reset when execution of a user action ends.

When a user action is executed by a rule firing, this variable will initially contain information specific to that rule. If not executed by a rule, this does not apply. The table below indicates the contents of this variable based on the rule that caused the action to fire.

Rule Firing Event	[Data11] Contents
Call	Empty.
ACD Agent Status Changed	Empty.
ACD Agent Help	Empty.
Account Code Entered	Empty.
Digits To Voice Mail	Empty.
Forward / Divert Status Changed	Empty.
Do Not Disturb Status Changed	Empty.

Note: If you subsequently call a command that sets the contents of this variable, the initial information pertaining to the rule will be lost.

14.156.51 Data2

Data2

This variable stores temporary data assigned using such commands as **DataSetStr** or **DataSetNum**. There are 11 such variables, which can each store data independently. The contents of this variable are reset when execution of a user action ends.

When a user action is executed by a rule firing, this variable will initially contain information specific to that rule. If not executed by a rule, this does not apply. The table below indicates the contents of this variable based on the rule that caused the action to fire.

Rule Firing Event	[Data2] Contents
Call	A string containing one of the following values: • The DNIS description for inbound external calls. • The trunk line description for external non- calls. • “Internal” for internal calls.
ACD Agent Status Changed	The ACD agent ID whose status has changed.
ACD Agent Help	
Account Code Entered	The trunk line of the external call that the account code was entered for.
Digits To Voice Mail	Empty.
Forward / Divert Status Changed	The extension device or group being diverted to.
Do Not Disturb Status Changed	

Note: If you subsequently call a command that sets the contents of this variable, the initial information pertaining to the rule will be lost.

14.156.52 Data3

Data3

This variable stores temporary data assigned using such commands as **DataSetStr** or **DataSetNum**. There are 11 such variables, which can each store data independently. The contents of this variable are reset when execution of a user action ends.

When a user action is executed by a rule firing, this variable will initially contain information specific to that rule. If not executed by a rule, this does not apply. The table below indicates the contents of this variable based on the rule that caused the action to fire.

Rule Firing Event	[Data3] Contents
Call	The received or digits for the call. If no was received for an inbound external call, this string contains “[No]”.
ACD Agent Status Changed	
ACD Agent Help	
Account Code Entered	Empty.
Digits To Voice Mail	Empty.
Forward / Divert Status Changed	Empty.
Do Not Disturb Status Changed	

Note: If you subsequently call a command that sets the contents of this variable, the initial information pertaining to the rule will be lost.

14.156.53 Data4

Data4

This variable stores temporary data assigned using such commands as **DataSetStr** or **DataSetNum**. There are 11 such variables, which can each store data independently. The contents of this variable are reset when execution of a user action ends.

When a user action is executed by a rule firing, this variable will initially contain information specific to that rule. If not executed by a rule, this does not apply. The table below indicates the contents of this variable based on the rule that caused the action to fire.

Rule Firing Event	[Data4] Contents
Call	The digits for an inbound call, or an empty string for non- calls.
ACD Agent Status Changed	
ACD Agent Help	Empty.
Account Code Entered	Empty.
Digits To Voice Mail	Empty.
Forward / Divert Status Changed	Empty.
Do Not Disturb Status Changed	Empty.

Note: If you subsequently call a command that sets the contents of this variable, the initial information pertaining to the rule will be lost.

14.156.54 Data5

Data5

This variable stores temporary data assigned using such commands as **DataSetStr** or **DataSetNum**. There are 11 such variables, which can each store data independently. The contents of this variable are reset when execution of a user action ends.

When a user action is executed by a rule firing, this variable will initially contain information specific to that rule. If not executed by a rule, this does not apply. The table below indicates the contents of this variable based on the rule that caused the action to fire.

Rule Firing Event	[Data5] Contents
Call	The unique serial number for the call. The serial number is an internally generated string that the assigns to each external call.
ACD Agent Status Changed	
ACD Agent Help	Empty.
Account Code Entered	Empty.
Digits To Voice Mail	Empty.
Forward / Divert Status Changed	Empty.
Do Not Disturb Status Changed	Empty.

Note: If you subsequently call a command that sets the contents of this variable, the initial information pertaining to the rule will be lost.

14.156.55 Data6

Data6

This variable stores temporary data assigned using such commands as **DataSetStr** or **DataSetNum**. There are 11 such variables, which can each store data independently. The contents of this variable are reset when execution of a user action ends.

When a user action is executed by a rule firing, this variable will initially contain information specific to that rule. If not executed by a rule, this does not apply. The table below indicates the contents of this variable based on the rule that caused the action to fire.

Rule Firing Event	[Data6] Contents
Call	This contains Field 2 from the matched record in the Telephone Import datat
ACD Agent Status Changed	Empty.
ACD Agent Help	Empty.
Account Code Entered	Empty.
Digits To Voice Mail	Empty.
Forward / Divert Status Changed	Empty.
Do Not Disturb Status Changed	Empty.

Note: If you subsequently call a command that sets the contents of this variable, the initial information pertaining to the rule will be lost.

14.156.56 Data7

Data7

This variable stores temporary data assigned using such commands as **DataSetStr** or **DataSetNum**. There are 11 such variables, which can each store data independently. The contents of this variable are reset when execution of a user action ends.

When a user action is executed by a rule firing, this variable will initially contain information specific to that rule. If not executed by a rule, this does not apply. The table below indicates the contents of this variable based on the rule that caused the action to fire.

Rule Firing Event	[Data7] Contents
Call	This contains Field 3 from the matched record in the Telephone Import datat
ACD Agent Status Changed	Empty.
ACD Agent Help	Empty.
Account Code Entered	Empty.
Digits To Voice Mail	Empty.
Forward / Divert Status Changed	Empty.
Do Not Disturb Status Changed	Empty.

Note: If you subsequently call a command that sets the contents of this variable, the initial information pertaining to the rule will be lost.

14.156.57 Data8

Data8

This variable stores temporary data assigned using such commands as **DataSetStr** or **DataSetNum**. There are 11 such variables, which can each store data independently. The contents of this variable are reset when execution of a user action ends.

When a user action is executed by a rule firing, this variable will initially contain information specific to that rule. If not executed by a rule, this does not apply. The table below indicates the contents of this variable based on the rule that caused the action to fire.

Rule Firing Event	[Data8] Contents
Call	This contains Field 4 from the matched record in the Telephone Import database.
ACD Agent Status Changed	Empty.
ACD Agent Help	Empty.
Account Code Entered	Empty.
Digits To Voice Mail	Empty.
Forward / Divert Status Changed	Empty.
Do Not Disturb Status Changed	Empty.

Note: If you subsequently call a command that sets the contents of this variable, the initial information pertaining to the rule will be lost.

14.156.58 Data9

Data9

This variable stores temporary data assigned using such commands as **DataSetStr** or **DataSetNum**. There are 11 such variables, which can each store data independently. The contents of this variable are reset when execution of a user action ends.

When a user action is executed by a rule firing, this variable will initially contain information specific to that rule. If not executed by a rule, this does not apply. The table below indicates the contents of this variable based on the rule that caused the action to fire.

Rule Firing Event	[Data9] Contents
Call	This contains Field 5 from the matched record in the Telephone Import datat
ACD Agent Status Changed	Empty.
ACD Agent Help	Empty.
Account Code Entered	Empty.
Digits To Voice Mail	Empty.
Forward / Divert Status Changed	Empty.
Do Not Disturb Status Changed	Empty.

Note: If you subsequently call a command that sets the contents of this variable, the initial information pertaining to the rule will be lost.

14.156.59 DDE1

DDE1

This variable represents the data returned by the last **DDERequest** command for DDE channel 1.

14.156.60 DDE2

DDE2

This variable represents the data returned by the last **DDERequest** command for DDE channel 2.

14.156.61 DDE3

DDE3

This variable represents the data returned by the last **DDERequest** command for DDE channel 3.

14.156.62 DDE4

DDE4

This variable represents the data returned by the last **DDERequest** command for DDE channel 4.

14.156.63 DDE5

DDE5

This variable represents the data returned by the last **DDERequest** command for DDE channel 5.

14.156.64 DDE6

DDE6

This variable represents the data returned by the last **DDERequest** command for DDE channel 6.

14.156.65 DDIDigits

DDIDigits

This variable equals the digits for the current call. If the call is not a call, the variable will contain an empty string.

For an action that is executed because a rule fired, the current call is the call in the call list that caused the rule to fire. If a rule did not execute the action, then the current call is the selected call in the call list.

You can select a particular call in the call list using the **CallSelect** command.

14.156.66 DevFirstRung

DevFirstRung

This variable equals the extension or group device that was rung first by the current call. For an inbound call this will be the first device that the call alerted. For an outbound external call on a trunk line, this variable will contain the trunk line that the outbound call is connected to. For an outbound internal call, it will contain the extension device of the party that the call was made from.

For an action that is executed because a rule fired, the current call is the call in the call list that caused the rule to fire. If a rule did not execute the action, then the current call is the selected call in the call list.

This variable will never return a device that is not entered into ’s Extension or Groups window.

You can select a particular call in the call list using the **CallSelect** command.

14.156.67 DialCombo

DialCombo

This variable equals the current value that is entered in the Dial Area of the CallViewer . This value is either the telephone number that the user last entered to dial, or the telephone number of the last call active at the extension.

14.156.68 DialLast

DialLast

This variable equals the first item in the CallViewer Dial List. The Dial List contains the last 20 telephone numbers that called this extension or were called by it. The first item in the Dial List is therefore the telephone number of the last person contacted.

14.156.69 DialPrefix

DialPrefix

This variable represents the dial prefix used when making outbound calls, as stored on the MiCC Office Server . If CallViewer is using local dial rules, this variable will still return the MiContact Center Office Server configured dial prefix.

This string will be blank if CallViewer is not connected to MiContact Center Office Server .

14.156.70 Digits

Digits

This variable contains the digits or received for the current call in the call list. If was not received for an inbound external call, this variable contains “[No]”.

For an action that is executed because a rule fired, the current call is the call in the call list that caused the rule to fire. If a rule did not execute the action, then the current call is the selected call in the call list. You can select a particular call in the call list using the **CallSelect** command.

This variable can be used to perform screen popping of an external application, whereby the caller’s details are displayed in the company database when a call is made or received.

However, performing a search by telephone number in a database can be inexact. This is because different users will enter telephone numbers in different formats, e.g. “ ”, while this variable contains an unformatted telephone number, e.g., “ .”

An alternative method of screen popping, which is more reliable, is to use the 6-field Telephone Import file with to import known contact information from the company database on a regular basis. Four of these six fields can contain custom information, which is made available in the [**Col4**], [**Col5**], [**Col6**], and [**Col7**] variables. For example, if [**Col7**] contained the primary key for the associated record in the company database, locating the correct record when a call is made or received could be achieved very quickly in a screen popping macro.

Note: If an e-mail has been routed to , you should use the [**EmailFromAddr**] variable for information on where it came from, because [**Digits**] applies only to call-based media.

14.156.71 DNIS

DNIS

This variable displays call information that changes depending on the type of the current call.

For inbound external calls it will contain the DNIS description associated with the number by the distant end.

For external non- calls it will contain the description of the trunk line that the call is on.

For internal calls it will contain “[Internal].”

For an action that is executed because a rule fired, the current call is the call in the call list that caused the rule to fire. If a rule did not execute the action, the current call is the selected call in the call list. You can select a particular call in the call list using the **CallSelect** command.

14.156.72 EmailFromAddr

EmailFromAddr

This variable contains the e-mail address of the original sender of an e-mail that has been routed to the agent logged in to the extension associated with this . If the agent has not been routed an e-mail, this variable is a blank string.

This variable can be used to perform screen popping of an external application, whereby the contact's details are displayed in the company database when an e-mail is received. This would be achieved by searching the company database for records matching the given e-mail address.

A more reliable method however, is to use the 6-field Telephone Import file with to import known contact information from the company database on a regular basis. Four of these six fields can contain custom information, which is made available in the [Col4], [Col5], [Col6], and [Col7] variables. For example, if [**Col7**] contained the primary key for the associated record in the company database, locating the correct record when an e-mail is received could be achieved very quickly in a screen popping macro.

14.156.73 EmailFromName

EmailFromName

This variable contains the description assigned against the e-mail address of the original sender of an e-mail that has been routed to the agent logged in to the extension associated with this CallViewer . The e-mail address description is often defined by the original sender of the e-mail, or by their local e-mail server. If the agent has not been routed an e-mail, this variable is a blank string.

14.156.74 EmailGrpQ

EmailGrpQ

This variable contains the device number of the hunt group that an e-mail arrived was routed from. This only applies for e-mails routed to the agent that is logged in at the extension that CallViewer is associated with. When such an e-mail is routed, the hunt group device corresponds to the media blending queue that the e-mail message arrived via. This is defined within the Group configuration of Contact Center Server . If the agent has not been routed an e-mail, this variable is a blank string.

14.156.75 EmailProcessing

EmailProcessing

This variable returns 1 if the extension associated with is processing an e-mail message routed by , otherwise it returns 0.

This variable is useful when writing user actions to perform screen popping using the or e-mail address that may need to deal with calls and e-mails being received at the same time. This variable can be used to determine whether to perform the screen pop using the / digits, or the e-mail address of the message originator.

If you your screen popping action uses one of the [**Col*x***] variables instead, along with the Telephone Import database, the [**EmailProcessing**] variable will be of use only if trying to decide if other e-mail related variables are valid or not.

14.156.76 EmailSize

EmailSize

This variable returns the size in bytes of the e-mail message that has been routed to the agent logged in at the extension associated with CallViewer . The size includes any message headers or attachments in the message as well. If the agent has not been routed an e-mail, this variable will return 0.

14.156.77 EmailSubjectText

EmailSubjectText

This variable returns the subject line of the e-mail message that has been routed to the agent logged in at the extension associated with CallViewer . The subject line is exactly the same as the subject line in the e-mail that is available in the agent's e-mail client. If the agent has not been routed an e-mail, this variable returns an empty string.

14.156.78 EmailTag

EmailTag

This variable returns the tag code of the e-mail message that has been routed to the agent logged in at the extension associated with CallViewer . The e-mail tag code is an internal reference string assigned by Contact Center Server to the e-mail message. It is unique in real-time for all active e-mail messages being queued by any instance of Intelligent Router . If the agent has not been routed an e-mail, this variable returns an empty string.

14.156.79 EmailTagOrig

EmailTagOrig

This variable returns the original tag number of the e-mail message that has been routed to the agent logged in at the extension associated with CallViewer . The original e-mail tag number is an internal reference of the e-mail messages assigned by Intelligent Router r instance that downloaded the original message.

The original tag is also incorporated into the subject text of the routed e-mail, and so can be viewed in the e-mail client that receives the routed messages. When incorporating the tag into the subject text, it is surrounded between “#[” and “]#” as well as being formatted in hexadecimal (base 16), and padded to 8 characters, so a tag of 19 would appear as “#[00000013]#”.

14.156.80 EmailToAddr

EmailToAddr

This variable returns the e-mail address of the mailbox that the external e-mail was sent to, for e-mails that have been routed to the agent logged in at the extension associated with CallViewer . This e-mail address equates to the address associated with a media blending queue. If the agent has not been routed an e-mail, this variable returns an empty string.

14.156.81 EmailToName

EmailToName

This variable contains the description assigned against the e-mail address of the mailbox that the external e-mail was sent to. This only applies to those e-mails that have been routed to the agent logged in to the extension associated with this CallViewer . The e-mail address description is often defined by the original sender of the e-mail, or by their local e-mail server. If the agent has not been routed an e-mail, this variable is a blank string.

14.156.82 EOF1

EOF1

This variable returns a non-zero value if the current file position related to file handle 1 is at the end of the file, e.g. having read every line in the file. If the return value is zero, the current file position is not at the end of the file.

14.156.83 EOF2

EOF2

This variable returns a non-zero value if the current file position related to file handle 2 is at the end of the file, e.g. having read every line in the file. If the return value is zero, the current file position is not at the end of the file.

14.156.84 EOF3

EOF3

This variable returns a non-zero value if the current file position related to file handle 3 is at the end of the file, e.g. having read every line in the file. If the return value is zero, the current file position is not at the end of the file.

14.156.85 EOF4

EOF4

This variable returns a non-zero value if the current file position related to file handle 4 is at the end of the file, e.g. having read every line in the file. If the return value is zero, the current file position is not at the end of the file.

14.156.86 EOF5

EOF5

This variable returns a non-zero value if the current file position related to file handle 5 is at the end of the file, e.g. having read every line in the file. If the return value is zero, the current file position is not at the end of the file.

14.156.87 ErrorDesc

ErrorDesc

This variable returns descriptive text for the error returned by the last DDExxx, Filexxx, or ODBCxxx command performed. This only applies if the SetErrorsFatal command has been used to stop errors from terminating execution of the macro.

14.156.88 ErrorNum

ErrorNum

This variable returns the error number for the error returned by the last DDExxx, Filexxx, or ODBCxxx command performed. This only applies if the SetErrorsFatal command has been used to stop errors from terminating execution of the macro.

14.156.89 INIFile

INIFile

This variable returns a string that contains the name of the initialization (INI) file where some settings are stored. This variable is provided for backward compatibility. In version 4 the vast majority of settings are stored in the registry, rather than in an INI file.

14.156.90 Line

Line

This variable returns the line or extension number for the current call. For an action that is executed because a rule fired, the current call is the call in the call list that caused the rule to fire. If a rule did not execute the action, then the current call is the selected call in the call list. You can select a particular call in the call list using the **CallSelect** command.

14.156.91 LocalExtension

LocalExtension

This variable returns the extension that CallViewer is associated with.

14.156.92 LongDate

LongDate

This variable returns a string representing the current date in “long” format, as defined in the Regional settings of Windows Control Panel. A long date format would be similar to “ April 22, 2006 ”.

14.156.93 LongDistPref

LongDistPref

This variable represents the long distance telephone number prefix used when making long distance calls, as stored on the MiContact Center Office Server . If CallViewer is using local dial rules, this variable will still return the MiCC Office Server configured long distance telephone number prefix.

This string will be blank if is not connected to MiCC Office Server .

14.156.94 LongTime

LongTime

This variable returns the current time in long format, as defined in the Regional settings of Windows Control Panel.

14.156.95 Macros

Macros

This variable returns the number of macros that are concurrently running. This is provided for backwards compatibility with earlier versions of where there was a limit on the number of macros that could run concurrently. In version 4, all user actions run independently of each other, and several can run at once. In version 4, this variable always returns 1.

14.156.96 MacrosNested

MacrosNested

This variable returns the nested level of this user action, which can be used to decide if the user action has been executed by another. This is provided for backwards compatibility with earlier versions of CallViewer . In version 4 all user actions run independently of each other, and so this variable always returns 0.

14.156.97 MediumDate

MediumDate

This variable returns the current date in “medium” date format. The medium date format is the same as the short date format, as defined by Regional settings in Windows Control Panel, except that the month is spelled out in abbreviated form rather than numeric form.

14.156.98 MediumTime

MediumTime

This variable returns the current time in 12-hour format using hours, minutes, and an AM/PM designator.

14.156.99 ODBCPos1

ODBCPos1

This variable returns the current record position for the open ODBC connection on ODBC channel 1. Since the ODBC driver may not be able to give reliable information on physical record position, this variable will only ever return one of the following values:

-1	At the end of the records (i.e. no more records left to process)
0	At the first record.
1	In the middle, i.e. not at the first or last record.

Note: You should use the GotoIfNoRecords command to check to see if any records have been returned, rather than using this macro variable.

14.156.100 ODBCPos2

ODBCPos2

This variable returns the current record position for the open ODBC connection on ODBC channel 2. Since the ODBC driver may not be able to give reliable information on physical record position, this variable will only ever return one of the following values:

-1	At the end of the records (i.e. no more records left to process)
0	At the first record.
1	In the middle, i.e. not at the first or last record.

Note: You should use the GotoIfNoRecords command to check to see if any records have been returned, rather than using this macro variable.

14.156.101 ODBCPos3

ODBCPos3

This variable returns the current record position for the open ODBC connection on ODBC channel 3. Since the ODBC driver may not be able to give reliable information on physical record position, this variable will only ever return one of the following values:

-1	At the end of the records (i.e. no more records left to process)
0	At the first record.
1	In the middle, i.e. not at the first or last record.

Note: You should use the GotoIfNoRecords command to check to see if any records have been returned, rather than using this macro variable.

14.156.102 RND

RND

This variable generates a random number between 0 and 1, e.g., 0.45125. The number is accurate to 5 decimal places.

14.156.103 ShortDate

ShortDate

This variable returns the current date in “short” date format, as defined in the Regional settings of Windows Control Panel.

14.156.104 ShortTime

ShortTime

This variable returns the current time in “short” time format, as defined in the Regional settings of Windows Control Panel.

14.156.105 TelNoFormatCount

TelNoFormatCount

This variable returns the number of telephone number formats that are recognized by the FormatTelephoneNumber command. If you use the FormatTelephoneNumber command, and request a format beyond the value of this variable, then the command will return an empty string.

14.156.106 Titlebar

Titlebar

This variable returns the titlebar text of the application window that is currently active. If there is no active window, this variable returns a blank string.

14.156.107 WinDir

WinDir

This variable returns a string containing the full path to the current Windows folder on this computer, e.g., `C:\Windows`. It is recommended that you use this variable rather than hard-coding the Windows folder directly, because the location of the folder could change between different computers or operating systems.

14.156.108 WinOS

WinOS

This variable returns a string that identifies the operating system that CallViewer is currently running on. It can be one of the following values:

Value	Operating System
WIN95	Windows 95
WIN98	Windows 98
WINME	Windows ME
WINNT	Windows NT
WIN2000	Windows 2000
WINXP	Windows XP up to and including Service Pack 1
WINXP_SP2	Windows XP Service Pack 2 or later
WINUNK	Unknown operating system

14.156.109 WinSysDir

WinSysDir

This variable returns a string containing the full path to the current Windows System folder on this computer, e.g., `C:\Windows\System32`. It is recommended that you use this variable rather than hard-coding the System folder directly, since the location of the folder could change between different computers or operating systems.

15 Methods

The section documents the methods that are provided via Callview Link Control. Examples in the section assume VBScript as a language, and that an instance of the control has been assigned to the axCallview variable.

15.1 AppActivateLike

AppActivateLike

This method activates a particular window. An activated window has the input focus, receiving all keystrokes, as well as being at the front of the desktop.

Syntax:

AppActivateLike WindowTitle

Parameter:

WindowTitle: The left part of the title of the application window to activate.

The name that appears in the titlebar need not be fully specified. For instance, “Calculat” would still activate an open application with titlebar text “Calculator”. The comparison is also not case sensitive; i.e., “Calculator” and “calculator” appear identical.

Example:

```
' Activate the window with a caption of “Untitled - Notepad”
axCallview.AppActivateLike “Untitled - Notepad”
' Activate a window whose caption starts with “Call”
axCallview.AppActivateLike “Call”
```

15.2 AppActivateLikeChild

AppActivateLikeChild

This method activates a particular child window within a particular application window. An activated window has the input focus, receiving all keystrokes, as well as being at the front of the desktop.

Syntax:

AppActivateLikeChild WindowTitle, ChildWindow

Parameters:

- **WindowTitle**: The left part of the title of the application window to activate.

  The name that appears in the titlebar need not be fully specified. For instance, “Calculat” would still activate an open application with titlebar text “Calculator”. The comparison is also not case sensitive; i.e., “Calculator” and “calculator” appear identical.

- **ChildWindow**: The leftmost part of the title of the child window to activate.

  The name that appears in the titlebar need not be fully specified, and is not case sensitive.

Example:

```
' Activate the child window “Document1” in Word
axCallview.AppActivateLikeChild “Microsoft Word”, “Document1”
```

15.3 CallAnswer

CallAnswer

This method attempts to answer a call ringing on a given extension.

Syntax:

CallAnswer Extension, CallItem

Parameters:

- **Extension**: The extension device to answer the call at. If a blank string is specified, then the extension assigned to the running instance of will be used.

- **CallItem**: The index of the call to answer. Calls in the list are identified as 1 for the first call in the call list, 2 for the second call etc. A value of 0 will instruct to automatically answer the first alerting call that it finds against the given extension. In fact, if the call specified in this argument is not alerting the given extension, then the next alerting call is answered instead.

Example:

```
' Answer the current call at this extension
axCallview.CallAnswer "", 0
' Answer second call at extension 200
axCallview.CallAnswer "200", 2
```

15.4 CallConference

CallConference

This method allows you to conference calls at the given device

The following rules apply to this method:

- If all the calls at the specified extension are not held, then the method places the current call on hold and prompts you to enter in the extension or telephone number of another party.
- If there are any held calls at the specified extension then the method joins all the calls together into a conference.
- If the extension device is in a state that cannot facilitate the conferencing of calls (or adding a new conference party), then an error occurs.

Syntax:

CallConference Extension

Parameter:

Extension: The extension device to conference calls at. If a blank string is specified, then the extension assigned to the running instance of will be used.

Example:

```
' Conference the call at the current extension
axCallview.CallConference “”
```

15.5 CallDialDigits

CallDialDigits

This method dials digits over the active conversation on a given extension.

Syntax:

CallDialDigits Extension, CallItem, Digits

Parameters:

- **Extension**: The extension device to dial the digits at. If a blank string is specified, then the extension assigned to the running instance of will be used.

- **CallItem**: The index of the call to dial digits on. Calls in the list are identified as 1 for the first call in the call list, 2 for the second call etc. A value of 0 will instruct to automatically dial digits on the first answered call that it finds against the given extension.

  In fact, if the call specified is not in the answered state at the given extension, the next answered call is used instead.

- **Digits**: The digits to dial on the specified call. Some characters have a special meaning :

Character	Description
!	This character can precede a feature code. By dialing features codes you can simulate an extens feature being “accessed” on a station device. You can usually do this even when the [CanCallDi] macro variable returns a value that indicates that dialing digits on line is unavailable. See your extension manual for a list of default feature code values.
P	Pause
F	Hookflash

15.6 CallDrop

CallDrop

This method releases a particular call on a given extension.

Syntax:

CallDrop Extension, CallItem

Parameters:

- **Extension**: The extension device to end the call at. If a blank string is specified, then the extension assigned to the running instance of will be used.

- **CallItem**: The index of the call to end. Calls in the list are identified as 1 for the first call in the call list, 2 for the second call etc. A value of 0 will instruct to automatically drop the first call at the given extension that is outbound external or answered.

  In fact, if the call specified at the given extension is not an outbound external call, or answered, the first such call at the extension is used instead.

Example:

```
' Drop the current call at this extension
axCallview.CallDrop "", 0
' Drop the second call at extension 200
axCallview.CallDrop "200", 2
```

15.7 CallDropAll

CallDropAll

This method releases all calls on a given extension, and then resets the extension.

Syntax:

CallDropAll Extension

Parameter:

Extension: The extension device to end all calls at. If a blank string is specified, the extension assigned to the running instance of will be used.

Example:

```
' Drop all calls at the current extension
axCallview.CallDropAll “”
' Drop all calls at extension 200
axCallview.CallDropAll “200”
```

15.8 CallHoldExclusive

CallHoldExclusive

This method places an external outbound or answered call on exclusive hold at the given extension.

Syntax:

CallHoldExclusive Extension, CallItem

Parameters:

- **Extension**: The extension device to exclusively hold the call at. If a blank string is specified, then the extension assigned to the running instance of will be used.

- **CallItem**: The index of the call to hold. Calls in the list are identified as 1 for the first call in the call list, 2 for the second call etc. A value of 0 will instruct to automatically hold the first call at the given extension that is outbound external or answered.

  In fact, if the call specified at the given extension is not an outbound external call, or answered, the next call at the extension is used instead.

Example:

```
' Hold the current call at this extension
axCallview.CallHoldExclusive "", 0
' Hold the third call at extension 230
axCallview.CallHoldExclusive "230", 3
```

15.9 CallHoldSystem

CallHoldSystem

This method places an external outbound or answered call on system hold, sometimes known as park, at the given extension.

Syntax:

CallHoldSystem Extension, CallItem

Parameters:

- **Extension**: The extension device to system hold (park) the call at. If a blank string is specified, then the extension assigned to the running instance of will be used.

- **CallItem**: The index of the call to hold. Calls in the list are identified as 1 for the first call in the call list, 2 for the second call etc. A value of 0 will instruct to automatically hold the first call at the given extension that is outbound external or answered.

  In fact, if the call specified at the given extension is not an outbound external call, or answered, the next call at the extension is used instead.

Example:

```
' Place the current call at this extension on system hold
axCallview.CallHoldSystem "", 0
' Hold the third call at extension 230
axCallview.CallHoldSystem "230", 3
```

15.10 CallMake

CallMake

This method attempts to make an outbound call from one extension to a given internal or external number.

Syntax:

CallMake Extension, DialString, AutoPrefix

Parameters:

- **Extension**: The extension device to make a new call at. If a blank string is specified, the extension assigned to the running instance of will be used.

- **DialString**: The telephone number to dial.

- **AutoPrefix**: This is a numerical argument that when set to “0” dials the digits exactly as entered in the “DialString” parameter.

  When this value is set to “1”, the number to be dialed is affected by the dial rules configured within CallViewer . At a minimum this means that the outbound dial prefix will be included in the digits sent to the telephone system.

Example:

```
' Make a call from the current extension to Mitel
axCallview.CallMake “”, “ 14809619000 ”, True
' Make a call from extension 213 to extension 200
axCallview.CallMake “213”, “200”, True
' Make a call to America without prefixing the dial prefix
axCallview.CallMake “”, “91410013104491481”, False
```

15.11 CallMonitor

CallMonitor

This method monitors an active call on an extension. The method returns True if successful, or False otherwise.

Syntax:

CallMonitor Extension, ExtTarget, MonitorType

Parameters:

- **Extension**: The extension device of the supervisor who will monitor the target call. If this is a blank string, the target call will be monitored at the extension currently assigned to CallViewer .
- **ExtTarget**: The extension where the external trunk line call is active. This call at this device will be monitored using the monitor type specified in the “MonitorType” parameter.
- **MonitorType**: This numerical value defines the type of monitoring to perform, as follows:

Value	Description
0	**Silent Monitor**: This allows an agent group supervisor to listen in on an agent's conversation from the supervisor extension. No indication is made to the agent or extension that is being monitored unless specified in the telephone system's programming.
1	Not used. : The target agent is informed of the intrusion by a beeping noise, and can refuse to accept the intrusic the agent accepts, the supervisor extension will intrude in on the conversation, as if they had been conferenced in.
2	Not used. : The target agent will be briefly informed of the intrusion by a beep, before the supervisor extension intrudes on the conversation, as if they had been conferenced in.

Example:

```
' Monitor a call at extension 216 in silent monitor mode
axCallview.CallMonitor “”, “216”, 0
' Monitor extension 213 at extension 278 in silent monitor mode
axCallview.CallMonitor "278", "213", 0
```

15.12 CallPage

CallPage

This method pages a group or extension from a given extension.

Syntax:

CallPage Extension, PageGroup

Parameters:

- **Extension**: The extension device to perform the page from. If this is a blank string, then the page will be performed at the extension currently assigned to CallViewer .
- **PageGroup**: The group to be paged.

Example:

```
' Page group 10 from the current extension
axCallview.CallPage "", "10"
```

15.13 CallPickup

CallPickup

This method picks up a call alerting at another extension, on a given extension.

Syntax:

CallPickup Extension, AlertingDevice

Parameters:

- **Extension**: The extension device to perform to pickup the call at. If this is a blank string, the call will be picked up at the extension currently assigned to CallViewer .
- **AlertingDevice**: The extension that has a queued or alerting call that is to be picked up at the “Extension” device.

Example:

```
' Pick up extension 213 at this extension
axCallview.CallPickup “”, “213”
' Pick up group 400 at extension 299
axCallview.CallPickup “299”, “400”
```

15.14 CallRetrieve

CallRetrieve

This method retrieves a call from exclusive hold, at a given extension.

Syntax:

CallRetrieve Extension, CallItem

Parameters:

- **Extension**: The extension device that has the held call to be retrieved. If this is a blank string, then the call will be recorded at the extension currently assigned to CallViewer .

- **CallItem**: The index of the call to retrieve. Calls in the list are identified as 1 for the first call in the call list, 2 for the second call etc. A value of 0 will instruct CallViewer to automatically retrieve the first exclusively held call at the given extension.

  In fact, if the call specified at the given extension is not an exclusively held call, then an error is generated.

Example:

```
' Retrieve the currently selected call on this extension
axCallview.CallRetrieve "", 0
' Retrieve the first call at extension "204"
axCallview.CallRetrieve "204", 1
```

15.15 CallSelect

CallSelect

This method selects a particular call in the call list associated with the CallViewer .

Syntax:

CallSelect CallItem

Parameter:

CallItem: The index of the call to select, where 1 represents the first call in the list, 2 represents the second call, and so on. Specifying a call item of 0 will remove any selection from the call list.

Example:

```
' Select the fourth call in the call list
axCallview.CallSelect 4
```

15.16 CallTransfer

CallTransfer

This method transfers the current call to a different number. It automatically places the currently answered call on exclusive hold, and then makes an announcement call to the new number.

Syntax:

CallTransfer Extension, DialString, AutoPrefix

Parameters:

- **Extension**: The extension to perform the transfer at. If the extension is a blank string, the device currently associated with is used instead.

- **DialString**: The telephone number to make the announcement call to. If this is a blank string then the user will be prompted for the number to transfer to.

- **AutoPrefix**: This is a numerical argument that when set to “0” dials the digits exactly as entered in the “DialString” parameter.

When this value is set to “1”, the number to be dialed is affected by the dial rules configured within CallViewer . At a minimum this means that the outbound dial prefix will be included in the digits sent to the telephone system (if the telephone number appears to be external).

Example:

```
' Transfer a call from the current extension to “200”
axCallview.CallTransfer “”, “200”, True
' Transfer a call from extension 200 to extension 416
axCallview.CallTransfer “200”, “416”, False
```

15.17 CallTransferComplete

CallTransferComplete

This command completes a call transfer at the given extension. For the command to be able to work there must already be the call-to-transfer on exclusive hold at the specified extension. There must also be a previously set up consultation call that is in the answered state. The command transfers the specified held call to the party at the distant end of the currently answered consultation call.

Syntax:

CallTransferComplete Extension, CallItem

Parameters:

- **Extension**: The extension device that has the consultation call to be completed. If this is a blank string, then the transfer will be completed at the extension currently assigned to CallViewer .

- **CallItem**: The index of the held call to transfer to the answered consultation call. Calls in the list are identified as 1 for the first call in the call list, 2 for the second call etc. A value of 0 will instruct to automatically transfer the first held call at the given extension.

Example:

```
' Complete a transfer of the currently selected call at the
' current extension
axCallview.CallTransferComplete "", 0
```

15.18 CallTransRedir

CallTransRedir

This command performs a blind (direct) transfer of an answered or alerting call at the given device to another party. The command prompts the user for the party to transfer the call to.

Syntax:

CallTransRedir Extension, CallItem

Parameters:

- **Extension**: The extension device that has the call to be blind transferred. If this is a blank string, then the transfer will be performed at the extension currently assigned to CallViewer .

- **CallItem**: The index of the call to blind transfer. Calls in the list are identified as 1 for the first call in the call list, 2 for the second call etc. A value of 0 will instruct to automatically transfer the first alerting, answered, or external outbound call at the given extension.

Example:

```
' Blind transfer the current call on the current extension
axCallview.CallTransRedir "", 0
```

15.19 CallTransRedirDirect

CallTransRedirDirect

This command performs a blind (direct) transfer of an answered or alerting call at the given device to another party specified in the command's parameters.

If the given call is alerting, it will be redirected to the specified party. If the call is answered, then it will be blind transferred. The call always alerts the party that it is transferred to immediately after the command is completed.

Syntax:

CallTransRedirDirect(Extension, CallItem, DialString)

Parameters:

- **Extension**: The extension device that has the call to be blind transferred. If this is a blank string, then the transfer will be performed at the extension currently assigned to CallViewer .
- **CallItem**: The index of the call to blind transfer. Calls in the list are identified as 1 for the first call in the call list, 2 for the second call etc. A value of 0 will instruct to automatically transfer the first alerting, answered, or external outbound call at the given extension.
- **DialString**: The number of the party to transfer the call to.

Example:

```
' Transfer a call from the current extension to extension 321
axCallview.callTransRedirDirect "", 0, "321"
```

15.20 DoCommand

DoCommand

This method executes any Macro Language command, or sequence of commands.

Syntax:

DoCommand(Command)

Parameter:

Command: A string that contains the Macro Language command(s) to perform. If more than one command is specified then each command be separated by a carriage return (ASCII character 10).

Example:

```
' Display a  message box
axCallview.DoCommand "MessageBox(""My message"",16,""My title"")"
```

15.21 GetDigitFormat

GetDigitFormat

This method returns the given continuous digit string as a telephone number. This is useful when searching a database for telephone numbers where a variety of different telephone number formats are used by the user, e.g., “ ” or “ .”

Syntax:

GetDigitFormat(Digits, FormatIndex) As String

Parameters:

- **Digits**: The continuous string of digits to be converted into a specific format. Typically this would be taken from the Digits property.

- **FormatIndex**: This number represents the index of the format to use. Indices are between 1 and the number returned by **DigitFormatCount**. Requesting an out of range index will return a blank string.

Example:

```
' Convert the digits in the Digits variable to the third format
szResult = axCallview.GetDigitFormat(axCallview.Digits, 3)
```

Notes:

The available number formats are stored in the “DialFormats” section of the file. The “FmtCount” setting indicates the number of available formats. For each format there is a setting “Fmt*XXX*” where *XXX* is the 1-based index of the format. Each format consists of hard coded text that will be output verbatim in the formatted string. The character “N” is used to represent the next digit in the input string, and the character “S” represents the rest of the entire string, including terminating any subsequent formatting.

For example, the format “+ (NNN) NS” would format “ ” as “ .”

15.22 GetSettingStr

GetSettingStr

This method reads a text setting from the **CVMACRO.INI** settings file. The method returns the setting stored against the given key in the requested section of the INI file, or returns the default value specified if no value could be found. The setting is returned as a text-based value.

Syntax:

GetSettingStr(Section, Setting, Default) As String

Parameters:

- **Section**: The section of the file where the settings can be found. A section is denoted by having its name enclosed in square brackets, e.g., “[Section Name]”
- **Setting**: The name of the setting to retrieve.
- **Default**: The default text to return if the given setting could not be found.

Example:

```
' Use the “ApplicationTitle” setting from the “General” section
' of the file in a message box
MsgBox “This is my test message”, 0, axCallview.GetSettingStr(“General”, “ApplicationTitle”, “”)
```

Notes:

The **CVMACRO.INI** file is broken down into sections and settings, e.g.,

```
[Section1]
Setting1=Value
Setting2=Other Value
```

15.23 GetSettingVal

GetSettingVal

This method reads a numeric setting from the **CVMACRO.INI** settings file. The method returns the setting stored against the given key in the requested section of the INI file, or returns the default value specified if no value could be found. The setting is returned as a numerical value.

Syntax:

GetSettingVal(Section, Setting, Default) As Long

Parameters:

- **Section**: The section of the file where the settings can be found. A section is denoted by having its name enclosed in square brackets, e.g., “[Section Name]”
- **Setting**: The name of the setting to retrieve.
- **Default**: The default value to return if the given setting could not be found.

Example:

```
' Initialize a variable with the “MaxTime” setting in the
' "Timers” section of the file
lMaxTimeSetting = axCallview.GetSettingStr(“Timers”, “MaxTime”, 30)
```

Notes:

The **CVMACRO.INI** file is broken down into sections and key entries, e.g.

[Section1]
Setting1=Value
Setting2=Other Value

15.24 Initialise

Initialise

This method must be called to enable the telephony events of the CallViewer Link Control. If the call is successful, the method returns True, and False if initialization failed. You only need call this method if you intend to process the events such as **CallAnswered**, etc. If you just want to use the CallViewer Link Control to perform some call control commands, you do not need to call this method.

Syntax:

Initialise

Note: This command uses British spelling and must be entered as shown here or it will produce a compiler error.

Parameters:

None.

Example:

```
If axCallview.Initialise then
       axCallview.CallMake "", " 1480 9619000 ", True
       axCallview.Uninitialise
End If
```

15.25 IsWindowOpen

IsWindowOpen

This method detects whether a specified window is open or not. It returns True if the window is open and visible, or False otherwise.

Syntax:

IsWindowOpen(WindowTitle) As Boolean

Parameter:

WindowTitle: The title of the window to find. This text can contain a number of “*s” as wildcards, representing zero or more characters. For example, “Calc*”, “*ulator”, and “*alcula*” would all match the “Calculator” title.

Example:

```
' Check if Microsoft Outlook is open
If axCallview.IsWindowOpen(“*Outlook”) then ...
' Check if Microsoft Word is open
If axCallview.IsWindowOpen(“Microsoft Word*”) then ...
' Check if a window starting ‘Microsoft’ & ending ‘Data’ is open
If axCallview.IsWindowOpen(“Microsoft*Data”) then ...
```

15.26 MacroBtnRun

MacroBtnRun

This method executes one of the 12 button macros imported from CallViewer version 3. This command only exists for support of existing version 3 macros, because it only executes macros that were created in version 3.x.

Syntax:

MacroBtnRun ButtonNumber

Parameter:

ButtonNumber: This numerical value depicts the button number from CallViewer version 3.x that should be executed. It is a value between 1 and 12.

Example:

```
' Run button macro 10
axCallview.MacroBtnRun 10
```

15.27 SendKeys

SendKeys

This method emulates sending a keystroke sequence to the currently active Windows-based application. Execution of the user action does not continue until the keystrokes have been processed.

Syntax:

SendKeys Keystrokes

Parameter:

Keystrokes: This string represents the keystrokes that are to be sent to the currently active application.

Most keystrokes are represented by each individual character in the string, for example the string “apple” would emulate pressing the “a” key, then “p,” “p,” “l,” and “e.”

You can emulate a key being pressed with Control, Shift, and/or Alt being pressed at the same time, with one of the following modifiers:

Character	Represents
Plus (+)	Shift key
Percent (%)	Alt key
Caret (^)	Control key

If you needed to emulate Control-Alt-X, you provide the text “^%x”. If you want to use the modifiers across several keys, enclose the keys in parentheses, e.g., “%(fa)” emulates Alt-F followed by Alt-A.

There are several keys that you cannot easily provide in the string, such as the Escape key, or the function keys. To emulate these keystrokes, use one of the following keywords in curly braces, e.g,. “{ENTER}”.

Keystroke	Keyword
Backspace	{BACKSPACE} or {BS} or {BKSP}
Break	{BREAK}
Caps Lock	{CAPSLOCK}
Clear	{CLEAR}
Del	{DELETE} or {DEL}
Down Arrow	{DOWN}
End	{END}
Enter	{ENTER} or ~ (tilde)
Esc	{ESCAPE} or {ESC}
Help	{HELP}
Home	{HOME}
Ins	{INSERT}
Left Arrow	{LEFT}
Num Lock	{NUMLOCK}
Page Down	{PGDN}

Page Up	{PGUP}
Right Arrow	{RIGHT}
Scroll Lock	{SCROLLLOCK}
Tab	{TAB}
Up Arrow	{UP}
F1	{F1}
F2	{F2}
F3	{F3}
F4	{F4}
F5	{F5}
F6	{F6}
F7	{F7}
F8	{F8}
F9	{F9}
F10	{F10}
F11	{F11}
F12	{F12}

If you need to emulate a key being pressed several times, then enclose the key or keyword in curly braces, followed by a space, and the number of times to repeat the key, e.g., “{ENTER 5}” would press the ENTER key five times.

If you need to type one of the special modifiers (%, ^, +, or ~), enclose them in curly braces too, e.g., “5% ” would press the 5 key followed by Alt+Space, whereas “5{%}” would type “5%”.

Example:

```
' Exit Notepad using the File menu
axCallview.AppActivateLike “Untitled – Notepad”
axCallview.Sendkeys “%fx”
```

15.28 SendKeysEx

SendKeysEx

This method emulates sending a keystroke sequence to the currently active application. Unlike the **SendKeys** method, **SendKeysEx** can be used to emulate keystrokes in both Windows applications, as well as MS-DOS and console application windows.

Syntax:

SendKeysEx Keystrokes, PauseTime

Parameters:

- **KeyStrokes**: This string represents the keystrokes that are to be sent to the currently active application. See the SendKeys method for information on the format of this string.

- **PauseTime**: This numerical value defines the number of milliseconds to pause between each keystroke. The value can be from 0 to 10000 (which is 10 seconds).

  It is recommended that a value other than 0 be used for this setting. This command is emulating keystrokes at the keyboard driver level, and while a human could not type 1000 keystrokes a second, **SendKeysEx** can! This can lead to the keyboard buffer being filled, and then keystrokes are lost. A value somewhere between 10 and 50 often works well, although experimentation with your chosen application is a good idea.

Example:

```
' Display help
axCallview.SendKeysEx "{F1}", 0
```

15.29 SetAccountCode

SetAccountCode

This method sets the account code for an external (trunk line) call at a given extension. The account code can either be written directly into the call model (allowing for alphanumeric account codes to be used), or via the telephone system. The method returns True if successful, or False otherwise.

Syntax:

SetAccountCode Extension, CallItem, AccountCode, UseCallControl

Parameters:

- **Extension**: The extension device to set the account code at. If a blank string is specified, the extension assigned to the running instance of will be used.

- **CallItem**: The index of the call to set the account code on. Calls in the list are identified as 1 for the first call in the call list, 2 for the second call etc. A value of 0 will instruct to automatically set the account code on the first external call at the given extension.

  In fact, if the call specified at the given extension is not an external call, the next call at the extension is used instead.

- **AccountCode**: The account code to set on the given call. If the “UseCallControl” parameter is 1, then this parameter has a maximum length of 12 characters, otherwise it has a maximum length of 50 characters.

  The account code in this parameter will overwrite any existing account code active on this call.

- **UseCallControl**: If this value is False then the account code bypasses the telephone system, and is processed only internally within Contact Center Server . This allows for longer account codes, and support for the account code functionality across all telephone systems.

  If the value is True, the account code is sent to the telephone system, as if it had been entered on the user’s handset. The account code is then limited to a length imposed by the telephone system.

Example:

```
' Set account code 999 on extension 214 via the telephone system
axCallview.SetAccountCode “214”, 0, “999”, True
' Set account code “NEW CUSTOMER” on current extension
axCallview.SetAccountCode “”, 0, “NEW CUSTOMER”, False
```

Notes:

The format of an account code varies by telephone system. However, by setting the UseCallControl flag to False, you can bypass the telephone system, and attach account codes to calls such that the account code can be alphanumeric, and up to 50 characters in length. This allows for greater flexibility when filtering with a product such as Reporter .

15.30 SetACDAgentState

SetACDAgentState

This method changes the state of an ACD agent, allowing the calling application to log an agent on or off, enter wrap-up, and so on. The method returns True if successful, or False otherwise.

Syntax:

SetACDAgentState(Extension, AgentID, AgentState, ACDGroup)

Parameters:

- **Extension**: The extension device to set the agent state at. If a blank string is specified, the extension assigned to the running instance of will be used.
- **AgentID**: The ID that defines the agent whose state is to change.
- **AgentState**: This numerical value depicts the new agent state to place the given extension / agent in. It can be one of the following values:

Value	Description
0	The agent is logged out. You can specify a specific hunt group to log out of in the “ACDGroup” parameter. If “ACDGroup” is blan agent is logged out of all hunt groups.
1	Log the agent in. In the “ACDGroup” parameter, you can specify a specific hunt group to log in to. If “ACDGroup” is blank agent is logged in to all hunt groups of which they are a member.
2	Changes the specified agent’s state to “Free.”
3	Changes the specified agent’s state to “Busy (Call).” **Note**: The “Busy (Call)” state is an automatic state generated when the corresponding agent is on a cal this reason, this command would normally generate an error if requesting that an agent enter the “Busy (Call)” state.
4	Changes the specified agent’s state to “Busy (E-mail).”
5	Changes the specified agent’s state to “Wrapup (Call).” This is performed in such a way that any teleph system-based wrap-up timer is ignored, and as such the agent will remain in the wrap-up state indefinit or at least until they enter the free state.
6	Changes the specified agent’s state to ” Do-Not-Disturb.”
7	Changes the specified agent's state to “Free (E-mail).”
8	Changes the specified agent's state to “Wrapup (E-mail).”

- **ACDGroup**: The ACD hunt group to log that agent in or out of. This only applies when logging in or out.

Example:

```
' Log agent 470 on at extension 200
axCallview.SetACDAgentState “200”, “470”, 1, “”
' Log the agent 470 off at this extension
axCallview.SetACDAgentState “”, “470”, 0, “”
' Put agent 470 on extension 213 into wrapup (call) state.
axCallview.SetACDAgentState “213”, “470”, 5, “”
```

15.31 SetSettingStr

SetSettingStr

This method writes a text-based setting to the **CVMACRO.INI** settings file.

Syntax:

SetSettingStr Section, Setting, Value

Parameters:

- **Section**: The section of the file where the settings can be found. A section is denoted by having its name enclosed in square brackets, e.g., “[Section Name]”
- **Setting**: The name of the setting to write.
- **Value**: The text value to write to this setting.

Example:

```
' Write the last number dialed to the “ LastDialed ” setting of
' the "Telephone" section
axCallview.SetSettingStr “Telephone”, “ LastDialed ”, axCallview.Digits
```

Notes:

The **CVMACRO.INI** file is broken down into sections and key entries, e.g.,

[Section1]
Setting1=Value
Setting2=Other Value

15.32 SetSettingVal

SetSettingVal

This method writes a numerical setting to the **CVMACRO.INI** settings file.

Syntax:

SetSettingVal Section, Setting, Value

Parameters:

- **Section**: The section of the file where the settings can be found. A section is denoted by having its name enclosed in square brackets, e.g., “[Section Name]”
- **Setting**: The name of the setting to write.
- **Value**: The numerical value to write to this setting.

Example:

```
' Write a 1 to the “MacroSuccessful” setting within the “General” section to indicate successful macro completion.
axCallview.SetSettingVal “General”, “MacroSuccessful”, 1
```

Notes:

The **CVMACRO.INI** file is broken down into sections and key entries, e.g.,

[Section1]

Setting1=Value

Setting2=Other Value

15.33 Shell

Shell

This command launches the given application, specified using its filename. If the file cannot be found, or a problem occurs launching the file, an error is generated.

Syntax:

Shell Filename

Parameter:

Filename: This string value is the fully pathed filename of the application to launch, e.g., C:\WINDOWS\NOTEPAD.EXE.

Example:

```
' Run Microsoft Access.
Shell("C:\MSOFFICE\ACCESS\MSACCESS.EXE")
```

15.34 ShellEx

ShellEx

This command launches an application with specific command line options. If the application file cannot be found, or a problem occurs launching the file, an error is generated.

Syntax:

ShellEx Filename, CommandLine

Parameters:

- **Filename**: This string value is the fully pathed filename of the application to launch, e.g., C:\WINDOWS\NOTEPAD.EXE.

- **CommandLine**: This string is the command line parameters to pass to the application. The contents of this string will depend on the application being launched.

Example:

```
' Run Microsoft Access and load the customer database.
axCallview.ShellEx "C:\ACCESS\MSACCESS.EXE", "C:\DATA\CUST.MDB"
```

15.35 Uninitialise

Uninitialise

This method should be called to disable the telephony events of the Link Control, having previously enabled the events with the **Initialise** method. The method returns True if events are successfully disabled and False otherwise.

Syntax:

Uninitialise

Parameters:

None.

16 Properties

This section documents the properties that are provided via the Callview Link Control.

16.1 AccountCode

AccountCode

This read-only property returns the account code last entered on the currently selected call.

16.2 ACDAgentID

ACDAgentID

This read-only property returns the Agent ID of the ACD Agent currently logged in to the extension associated with CallViewer .

16.3 ACDLoginCnt

ACDLoginCnt

This read-only property returns the number of times that the extension associated with CallViewer has logged into a “non-agent ID” type hunt group.

16.4 ACDLoginCntAgID

ACDLoginCntAgID

This read-only property returns the number of times that an ACD agent has logged into an “agent ID” type hunt group at the extension device that CallViewer is associated with in the current session.

16.5 ACDStatus

ACDStatus

This read-only property returns the current status of the ACD Agent logged in to the extension associated with CallViewer . The status is returned as an integer, with the following meaning:

Value	Description
0	Logged Out
1	Logged In
2	Free
3	Busy (Call)
4	Busy (E-mail)
5	Wrapup (Call)
6	Busy N/A (DND)
7	Wrapup (E-mail)
8	Free (E-mail)

16.6 CallAns

CallAns

This read-only property returns True if the currently selected call has been answered, and False otherwise.

16.7 CallAnsTime

CallAnsTime

This read-only property returns the date and time that the currently selected call was answered. The value is returned as a string in long date format, e.g., “ April 22, 2006 11:50:08”. The long date format is defined in the Regional settings section of the Windows Control Panel.

16.8 CallCLI

CallCLI

This read-only property returns True if the currently selected call was received with Caller ID and False otherwise.

16.9 CallContact

CallContact

This read-only property returns True if the currently selected call was identified by the MiCC Office Server using its Telephone Number Import. It returns False if the call was not identified, or if no call is present at the extension. Calls that were not received with Caller ID will always return False for this variable.

16.10 CallCtrl

CallCtrl

This read-only property returns True if has call control enabled, or False otherwise.

Call control can be enabled or disabled from the Enable Call Control option on the Call Control tab of the Options dialog. If the setting is disabled, no call control capability is available in CallViewer , either from the Macro Language or from the user interface.

16.11 CallHeld

CallHeld

This read-only property returns True if the currently selected call is currently held, or False otherwise.

16.12 CallId

CallId

This read only property returns the internal Call ID used by Server to identify a call. This can be used to index the calls in an application's own internal call model.

16.13 CallInt

CallInt

This read-only property returns True if the currently selected call is internal, or False for external.

16.14 CallMediaType

CallMediaType

This read-only property returns a string identifying the type of media that is currently being processed by CallViewer t. This property will only return a valid value if called within a user action executed by a rule firing.

Possible values are “CALL” if the user action was executed because of a call, or “EMAIL” if the action fired because of an e-mail being routed to the agent logged in at the extension associated with CallViewer .

16.15 CallOut

CallOut

This read-only property returns True if the currently selected call is outbound, or False for inbound.

The direction of the call is determined to how the call started on the current extension. For example, an external call may have been dialed to an external party, but subsequently transferred to another extension; in such an instance the call is considered inbound for the receiving extension, since it received a call, rather than made a call.

16.16 CallRingTime

CallRingTime

This read-only property returns the number of seconds that the currently selected call has been ringing for. If the call has been answered, this property will return the number of seconds that the call was ringing for.

16.17 Calls

Calls

This read-only property returns the number of calls active at the associated extension, or 0 if no calls are active.

16.18 CallSelected

CallSelected

This read-only property returns the index of the currently selected call in CallViewer ’s call list, or 0 if no calls are currently selected. The first call in the list is “1,” the next “2,” and so on.

16.19 CallSerialNo

CallSerialNo

This read-only property returns the internally generated unique serial number associated with the current call.

You can use call serial number to tag records in your database, and subsequently map your database records to call log information in the MiCC Office Server databases. The serial number is written to the “TTSerialNo” field in the MiCC Office Server databases.

16.20 CallSource

CallSource

This read-only property returns the index of the currently selected call in CallViewer ’s call list, or 0 if no calls are currently selected.

16.21 CallStartTime

CallStartTime

This read-only property returns the date and time that the currently selected call started to ring. The value is returned as a string in long date format, e.g., “ April 22, 2006 11:50:02”. The long date format is defined in the Regional settings of the Windows Control Panel.

16.22 CallWasOnHold

CallWasOnHold

This read-only property returns True if the current call was on hold, but was then retrieved, or False otherwise.

16.23 CanCallAnswer

CanCallAnswer

This read-only property returns True if a call active at the extension can currently be answered by CallViewer , or False otherwise.

Typically, answering is available if the following conditions are met:

- There is a call alerting the extension associated with CallViewer .
- Call control is enabled in CallViewer ’s options.
- The telephone driver supports answering calls via call control.
- The license obtained by allows calls to be answered.

16.24 CanCallConf

CanCallConf

This read-only property returns True if a call present at the extension can currently be conferenced by CallViewer , or False otherwise.

Typically, conference is available if the following conditions are met:

- There are answered or held calls at the extension associated with CallViewer .
- Call control is enabled in CallViewer ’s options.
- The telephone driver supports conferencing calls via call control.
- The license obtained by allows calls to be conferenced.

16.25 CanCallDial

CanCallDial

This read-only property returns True if new calls can be made by at the extension, or False otherwise.
Typically, dialing is available if the following conditions are met:

- The extension associated with is idle, or has an exclusively held call present.
- Call control is enabled in CallViewer ’s options.
- The telephone driver supports making calls via call control.
- The license obtained by allows calls to be dialed .

16.26 CanCallDialDig

CanCallDialDig

This read-only property returns True if digits can currently be dialed online at the extension by CallViewer , or False otherwise. Typically, dialing of digits is available if the following conditions are met:

- There is an answered call at the extension associated with CallViewer .
- Call control is enabled in CallViewer ’s options.
- The telephone driver supports dialing digits via call control.
- The license obtained by allows digits to be dialed .

16.27 CanCallDrop

CanCallDrop

This read-only property returns True if a call on the extension can currently be dropped by CallViewer , or False otherwise. Typically, dropping calls is available if the following conditions are met:

- There is an outbound or answered call at the extension associated with CallViewer .
- Call control is enabled in CallViewer ’s options.
- The telephone driver supports dropping calls via call control.
- The license obtained by allows calls to be dropped.

16.28 CanCallDropAll

CanCallDropAll

This read-only property returns True if the extension can currently be reset (all calls dropped) by CallViewer , or False otherwise.

Typically, dropping all calls is available if the following conditions are met:

- Call control is enabled in CallViewer ’s options.
- The telephone driver supports the handset being reset via call control.
- The license obtained by allows the handset to be reset via call control.

16.29 CanCallHoldEx

CanCallHoldEx

This read-only property returns True if the call currently at the extension can be exclusively held by CallViewer , or False otherwise.

Typically, exclusively holding calls is available if the following conditions are met:

- There is an external outbound, or answered call at the extension associated with CallViewer .
- Call control is enabled in CallViewer ’s options.
- The telephone driver supports exclusively holding calls via call control.
- The license obtained by allows calls to be held exclusively.

16.30 CanCallHoldSys

CanCallHoldSys

This read-only property returns True if the call currently on the extension can be placed on system hold (parked), or False otherwise.

Typically, system-holding calls is available if the following conditions are met:

- There is an external outbound, or answered call at the extension associated with CallViewer .
- Call control is enabled in CallViewer ’s options.
- The telephone driver supports system holding calls via call control.
- The license obtained by allows calls to be system held.

16.31 CanCallRetrieve

CanCallRetrieve

This read-only property returns True if the call currently selected can be retrieved from exclusive hold by CallViewer , or False otherwise.

Typically, retrieving calls is available if the following conditions are met:

- There is an exclusively held call at the extension associated with CallViewer .
- Call control is enabled in CallViewer ’s options.
- The telephone driver supports retrieving calls via call control.
- The license obtained by allows calls to be retrieved.

16.32 CanCallTrans

CanCallTrans

This read-only property returns True if an enquiry transfer can be performed on the currently selected call by CallViewer , or False otherwise.

Typically, enquiry transfer is available if the following conditions are met:

- There is an external outbound, or answered call at the extension associated with CallViewer .
- Call control is enabled in CallViewer ’s options.
- The telephone driver supports enquiry transfer of calls via call control.
- The license obtained by CallViewer allows for enquiry transfer of calls.

16.33 CanCallTransComp

CanCallTransComp

This read-only property returns True if the Complete Transfer call feature is enabled, or False otherwise.

Typically, complete transfer is available if the following conditions are met:

- There is a previously set-up consultation call at the extension associated with CallViewer , which is external outbound, or answered.
- Call control is enabled in CallViewer ’s options.
- The telephone driver supports complete transfer via call control.
- The license obtained by allows for transfer completion of calls.

16.34 CanCallTransRedir

CanCallTransRedir

This read-only property returns True if the Transfer / Redirect call control feature is enabled, or False otherwise. Typically, transfer/redirection of calls is available if the following conditions are met:

- There is an external or answered call at the extension associated with CallViewer .
- Call control is enabled in CallViewer ’s options.
- The telephone driver supports transferring / redirecting calls via call control.
- The license obtained by CallViewer allows calls to be transferred / redirected.

16.35 ClientActive

ClientActive

This read-only property returns True if the associated has a session with the MiCC Office Server , or False otherwise. If no session is established, telephone events, and call control methods will not work.

16.36 ClientName

ClientName

This read-only property returns the local network name associated with the CallViewer .

It is provided for backward compatibility with earlier versions of CallViewer , which used this value as the network name for the running instance of CallViewer .

16.37 ClientNameNum

ClientNameNum

This read-only property returns the local network name associated with CallViewer , with all non-numeric characters removed. This property is provided for backward compatibility to previous versions of CallViewer .

16.38 Clipboard

Clipboard

This read-only property returns the current contents of the clipboard, if the contents can be formatted as a text string. If the contents cannot be formatted as text, which is dependent on the application that put the text on the clipboard, then a blank string is returned.

16.39 Col(x)

Col(x)

This read-only property returns one of the seven **[Colx**] macro variables used by the Macro language.

- **Col(1)** returns the line or extension number for the current call.
- **Col(2)** returns either the DNIS description for direct dialed external calls, the trunk line description for non-direct dialed external calls, or the string “[Internal]” for internal calls.
- **Col(3)** to **Col(7)** return the corresponding fields from the Telephone Number Import on the Contact Center Server for the current call. Typically, **Col(3)** will equate to the caller’s company or name if it is identified against the Telephone Number import.

16.40 ConfPartyLimit

ConfPartyLimit

This read-only property returns the number of parties that can be included in a conference call. On some systems this will include the person who initiated the conference call. It is also possible that this property will return –1, if the maximum number is not known.

16.41 CTIServerName

CTIServerName

This read-only property returns the name of the Contact Center Server that the associated is connected to.

16.42 Data(x)

Data(x)

This read-only property returns the value associated with one of the eleven [**Data*x***] macro variables used by the Macro language.

16.43 DDE(x)

DDE(x)

This read-only property returns the value associated with one of the six [**DDE*x***] macro variables used the Macro language. The variable represents the data returned by the last **DDERequest** command for the given DDE channel.

16.44 DDIDigits

DDIDigits

This read-only property returns the DID Digits associated with the currently selected call.

16.45 DevFirstRung

DevFirstRung

This read-only property equals the extension or group device that was rung first by the current call. For an inbound call this will be the first device that the call alerted. For an outbound external call on a trunk line, this variable will contain the trunk line that the external call is connected to. For an outbound internal call, it will contain the extension device of the party that the call was made to.

This property will never return a device that is not entered into MiCC Office Server ’s Extension or Groups window.

16.46 DialCombo

DialCombo

This read-only property returns the current value that is entered in the Dial Area of the CallViewer . This value is either the telephone number that the user last entered to dial, or the telephone number of the last call active at the extension.

16.47 DialLast

DialLast

This read-only property returns the first item in the Dial List. The Dial List contains the last 20 telephone numbers that called this extension or were called by it. The first item in the Dial List is therefore the telephone number of the last person contacted.

16.48 DialPrefix

DialPrefix

This read-only property returns the dial prefix to use when making outbound calls, as configured in the MiCC Office Server . If is using local dial rules, this variable will still return the MiCC Office Server configured dial prefix.

This string will be blank if is not connected to MiCC Office Server .

16.49 DigitFormatCount

DigitFormatCount

This read-only property returns the number of formats that the **GetDigitFormat** method can support.

16.50 Digits

Digits

This read-only property returns the digits or the telephone number of the call currently at the associated .

This variable can be used to perform screen popping of an external application, whereby the caller's details are displayed in the company database when a call is made or received.

However, performing a search by telephone number in a database can be inexact. This is because different users will enter telephone numbers in different formats, e.g. " ", while this variable contains an unformatted telephone number, e.g., " ."

An alternative method of screen popping, which is more reliable, is to use the 6-field Telephone Import file with to import known contact information from the company database on a regular basis. Four of these six fields can contain custom information, which is made available in the fields 4 to 7 of the Col(x) property. For example, if **Col(7)** contained the primary key for the associated record in the company database, then locating the correct record when a call is made or received could be achieved very quickly in a screen popping macro.

Note: If an e-mail has been routed to , you should use the [**EmailFromAddr**] variable for information on where it came from, because [**Digits**] applies only to call-based media.

16.51 DNIS

DNIS

This read-only property returns one of the following string values for the current call:

- For inbound external DID calls it will contain the DNIS description associated with the DID number dialed by the distant end.
- For external non- DID calls it will contain the description of the trunk line that the call is on.
- For internal calls it will contain “[Internal].”

16.52 EmailFromAddr

EmailFromAddr

This read-only property contains the e-mail address of the original sender of an e-mail that has been routed to the agent logged in to the extension associated with this CallViewer . If the agent has not been routed an e-mail, then this property is a blank string.

This property can be used to perform screen popping of an external application, whereby the contact's details are displayed in the company database when an e-mail is received. This would be achieved by searching the company database for records matching the given e-mail address.

A more reliable method however, is to use the 6-field Telephone Import file with MiCC Office Server to import known contact information from the company database on a regular basis. Four of these six fields can contain custom information, which is made available in the fields 4 to 7 of the Col(x) property. For example, if **Col(7)** contained the primary key for the associated record in the company database, then locating the correct record when a routed e-mail is received could be achieved very quickly in a screen popping macro.

16.53 EmailFromName

EmailFromName

This read-only property contains the description assigned against the e-mail address of the original sender of an e-mail that has been routed to the agent logged in to the extension associated with this CallViewer .

The e-mail address description is often defined by the original sender of the e-mail, or by their local e-mail server. If the agent has not been routed an e-mail, this variable is a blank string.

16.54 EmailGrpQ

EmailGrpQ

This read-only property returns the device number of the group that this e-mail was sent to by the external party. For example, if the external party sends an e-mail to “sales@xyz.com,” and group “1000” is mapped to “sales@xyz.com,” this field will return “1000” if the e-mail sent to “sales@xyz.com” was routed by Intelligent Router r to the agent logged in at the extension that this is associated with.

16.55 EmailProcessing

EmailProcessing

This read-only property returns True if the extension associated with is processing an e-mail message routed by Intelligent Router , otherwise it returns False.

This variable is useful when writing user actions to perform screen popping using the Caller ID or e-mail address that may need to deal with calls and e-mails being received at the same time. This variable can be used to determine whether to perform the screen pop using the Caller ID / dialed digits, or the e-mail address of the message originator.

If you your screen popping action uses the **Col(x)** property instead, along with the Contact Center Server Telephone Import database, the **EmailProcessing** property will be of use only if trying to decide if other e-mail related variables are valid or not.

16.56 EmailSize

EmailSize

This read-only property returns the size in bytes of the e-mail that has been routed to the agent logged in at the extension that CallViewer is associated with. If the agent has not been routed an e-mail, this variable returns an empty string.

16.57 EmailSubjectText

EmailSubjectText

This read-only property returns the subject line of the e-mail message that has been routed to the agent logged in at the extension associated with CallViewer . The subject line is exactly the same as the subject line in the e-mail that is available in the agent’s e-mail client. If the agent has not been routed an e-mail, this property returns an empty string.

16.58 EmailTag

EmailTag

This read-only property returns the tag code of the e-mail message that has been routed to the agent logged in at the extension associated with CallViewer . The e-mail tag code is an internal reference string assigned by MiCC Office Server to the e-mail message. It is unique in real-time for all active e-mail messages being queued by any instance of Intelligent Router . If the agent has not been routed an e-mail, this property returns an empty string.

16.59 EmailTagOrig

EmailTagOrig

This read-only property returns the original tag number of the e-mail message that has been routed to the agent logged in at the extension associated with CallViewer . The original e-mail tag number is an internal reference of the e-mail messages assigned by Intelligent Router instance that downloaded the original message.

The original tag is also incorporated into the subject text of the routed e-mail, and so can be viewed in the e-mail client that receives the routed messages. When incorporating the tag into the subject text, it is surrounded between “#[” and “]#” as well as being formatted in hexadecimal (base 16), and padded to 8 characters, so a tag of 19 would appear as “#[00000013]#”.

16.60 EmailToAddr

EmailToAddr

This read-only property returns the e-mail address of the mailbox that the external e-mail was sent to, for e-mails that have been routed to the agent logged in at the extension associated with CallViewer . This e-mail address equates to the address associated with a media blending queue. If the agent has not been routed an e-mail, this property returns an empty string.

16.61 EmailToName

EmailToName

This read-only property contains the description assigned against the e-mail address of the mailbox that the external e-mail was sent to. This only applies to those e-mails that have been routed to the agent logged in to the extension associated with this CallViewer . The e-mail address description is often defined by the original sender of the e-mail, or by their local e-mail server. If the agent has not been routed an e-mail, this property is a blank string.

16.62 INIFile

INIFile

This read-only property returns the current INI file used by the associated CallViewer . This property is provided for backward compatibility. In version 4 the vast majority of settings are stored in the registry, rather than in an INI file.

16.63 IsConnected

IsConnected

This read-only property returns True if is currently running, and a successful connection is established between and the CallViewer Link Control. When this property returns False, it is likely that other property requests and other method calls will fail.

16.64 Line

Line

This read-only property returns the line or extension number for the current call in the associated call list.

16.65 LocalExtension

LocalExtension

This read-only property returns the actual extension number associated with the CallViewer .

16.66 LongDate

LongDate

This read-only property returns today’s date in a long format, as defined in the Regional settings of Windows Control Panel.

16.67 LongTime

LongTime

This read-only property returns the current time in long format, as defined in the Regional settings of Windows Control Panel.

16.68 Macros

Macros

This read-only property returns the number of macros that are concurrently running. This is provided for backwards compatibility with earlier versions of where there was a limit on the number of macros that could run concurrently. In version 4, all user actions run independently of each other, and several can run at once. In version 4, this variable always returns 1.

16.69 MacrosNested

MacrosNested

This variable returns the nested level of this user action, which can be used to decide if the user action has been executed by another. This is provided for backwards compatibility with earlier versions of CallViewer . In version 4 all user actions run independently of each other, and so this variable always returns 0.

16.70 MediumDate

MediumDate

This read-only property returns today’s date in medium format, as defined in the Regional settings of Windows Control Panel.

16.71 MediumTime

MediumTime

This read-only property returns the current time in medium format, as defined in the Regional settings of Windows Control Panel.

16.72 ShortDate

ShortDate

This read-only property returns today’s date in short format, as defined in the Regional settings of Windows Control Panel.

16.73 ShortTime

ShortTime

This read-only property returns the current time in short format, as defined in the Regional settings of Windows Control Panel.

16.74 Titlebar

Titlebar

This read-only property returns the titlebar text of the currently active application window.

16.75 Username

Username

This read-only property returns the name of the user currently logged in at this computer. If your computer does not require you to log on, this property will return a blank string.

16.76 WinDir

WinDir

This read-only property returns the full path to the Windows folder.

16.77 WinOS

WinOS

This read-only property returns the operating system that the script is running on. It can be one of the following:

Value	Operating System
WIN95	Windows 95
WIN98	Windows 98
WINME	Windows ME
WINNT	Windows NT
WIN2000	Windows 2000
WINXP	Windows XP up to and including Service Pack 1
WINXP_SP2	Windows XP Service Pack 2 or later
WINUNK	Unknown operating system

16.78 WinSysDir

WinSysDir

This read-only property returns the full path to the Windows system folder.

17 Events

This section describes the events that are provided via the Callview Link Control when it is enabled for events. Events are only fired by the control after the Initialise method has been called.

17.1 Busy

Busy

This event is fired when the handset is placed off hook, for example before dialing a phone number when the handset is picked up.

Parameters:

None.

17.2 CallAnswer

CallAnswer

This event is fired when a call is answered at the current extension. It is also fired when an e-mail is routed to the extension.

Parameters:

- **DN**: For external calls, this is the line that the call is on. For internal calls it is the extension called or calling.
- **DNIS**: For inbound direct dialed external calls, this is the DNIS text of the DID number. For other external calls, this is the description of the line that the call is on. For internal calls, the text [Internal] is used.
- **Caller ID** : For inbound external calls, this is the number of the calling party if the information was made available or the string [No Caller ID received!] if no Caller ID was provided. For all other calls, it is the number or extension that the call is connected to.
- **DID** : For direct dialed inbound external calls, this is the direct dial digits entered by the caller. The format of this string is telephone system dependent, but is usually either the pertinent section of the dialed number, or the entire dialeddialled number.
- **Serial Number**: For external calls, this is a unique string identifying the call. The MiCC Office Server uses this text to identify the call.
- **Call Flags**: This bitfield holds various information on the state of the call, as follows:

Value	Description
1	Internal call
2	Outbound call
4	Call held
8	Call answered
16	Contact identified in Telephone Import database
32	Caller ID received

- **Field 2**: This is the information from the second field of the Telephone Number import, if the contact was identified. Usually it is the contact's name or company name.
- **Field 3**: This is the information from the third field of the Telephone Number import, if the contact was identified. It is user-definable.
- **Field 4**: This is the information from the fourth field of the Telephone Number import, if the contact was identified. It is user-definable.
- **Field 5**: This is the information from the fifth field of the Telephone Number import, if the contact was identified. It is user-definable.
- **Field 6**: This is the information from the sixth field of the Telephone Number import, if the contact was identified. It is user-definable.

17.3 CallDigits

CallDigits

This event is fired when a call at the current extension receives a different Caller ID from that which was available when the call was first presented. The event is not fired when the call is first presented; it is only fired if the dialed digits change during a call.

Parameters:

- **DN**: For external calls, this is the line that the call is on. For internal calls it is the extension called or calling.
- **DNIS**: For inbound direct dialed external calls, this is the DNIS text of the DID number. For other external calls, this is the description of the line that the call is on. For internal calls, the text [Internal] is used.
- **Caller ID** : For inbound external calls, this is the number of the calling party if the information was made available or the string [No Caller ID received!] if no Caller ID was provided. For all other calls, it is the number or extension that the call is connected to.
- **DID** : For direct dialed inbound external calls, this is the direct dial digits entered by the caller. The format of this string is telephone system dependent, but is usually either the pertinent section of the dialed number, or the entire dialed number.
- **Serial Number**: For external calls, this is a unique string identifying the call. The MiCC Office Server uses this text to identify the call.
- **Call Flags**: This bitfield holds various information on the state of the call, as follows:

Value	Description
1	Internal call
2	Outbound call
4	Call held
8	Call answered
16	Contact identified in Telephone Import database
32	Caller ID received

- **Field 2**: This is the information from the second field of the Telephone Number import, if the contact was identified. Usually it is the contact's name or company name.
- **Field 3**: This is the information from the third field of the Telephone Number import, if the contact was identified. It is user-definable.
- **Field 4**: This is the information from the fourth field of the Telephone Number import, if the contact was identified. It is user-definable.
- **Field 5**: This is the information from the fifth field of the Telephone Number import, if the contact was identified. It is user-definable.
- **Field 6**: This is the information from the sixth field of the Telephone Number import, if the contact was identified. It is user-definable.

17.4 CallHeld

CallHeld

This event is fired when a call is put on either system or exclusive hold at the current extension.

Parameters:

- **DN**: For external calls, this is the line that the call is on. For internal calls it is the extension called or calling.
- **DNIS**: For inbound direct dialed external calls, this is the DNIS text of the DID number. For other external calls, this is the description of the line that the call is on. For internal calls, the text [Internal] is used.
- **Caller ID** : For inbound external calls, this is the number of the calling party if the information was made available or the string [No Caller ID received!] if no Caller ID was provided. For all other calls, it is the number or extension that the call is connected to.
- **DID** : For direct dialed inbound external calls, this is the direct dial digits entered by the caller. The format of this string is telephone system dependent, but is usually either the pertinent section of the dialed number, or the entire dialed number.
- **Serial Number**: For external calls, this is a unique string identifying the call. The MiCC Office Server uses this text to identify the call.
- **Call Flags**: This bitfield holds various information on the state of the call, as follows:

Value	Description
1	Internal call
2	Outbound call
4	Call held
8	Call answered
16	Contact identified in Telephone Import database
32	Caller ID received

- **Field 2**: This is the information from the second field of the Telephone Number import, if the contact was identified. Usually it is the contact's name or company name.
- **Field 3**: This is the information from the third field of the Telephone Number import, if the contact was identified. It is user-definable.
- **Field 4**: This is the information from the fourth field of the Telephone Number import, if the contact was identified. It is user-definable.
- **Field 5**: This is the information from the fifth field of the Telephone Number import, if the contact was identified. It is user-definable.
- **Field 6**: This is the information from the sixth field of the Telephone Number import, if the contact was identified. It is user-definable.

17.5 CallIdentified

CallIdentified

This event is fired when a call at the current extension is identified in the MiCC Office Server import.

Parameters:

- **DN**: For external calls, this is the line that the call is on. For internal calls it is the extension called or calling.
- **DNIS**: For inbound direct dialed external calls, this is the DNIS text of the DID number. For other external calls, this is the description of the line that the call is on. For internal calls, the text [Internal] is used.
- **Caller ID** : For inbound external calls, this is the number of the calling party if the information was made available or the string [No Caller ID received!] if no Caller ID was provided. For all other calls, it is the number or extension that the call is connected to.
- **DID** : For direct dialed inbound external calls, this is the direct dial digits entered by the caller. The format of this string is telephone system dependent, but is usually either the pertinent section of the dialed number, or the entire dialed number.
- **Serial Number**: For external calls, this is a unique string identifying the call. The MiCC Office Server uses this text to identify the call.
- **Call Flags**: This bitfield holds various information on the state of the call, as follows:

Value	Description
1	Internal call
2	Outbound call
4	Call held
8	Call answered
16	Contact identified in Telephone Import database
32	Caller ID received

- **Field 2**: This is the information from the second field of the Telephone Number import, if the contact was identified. Usually it is the contact's name or company name.
- **Field 3**: This is the information from the third field of the Telephone Number import, if the contact was identified. It is user-definable.
- **Field 4**: This is the information from the fourth field of the Telephone Number import, if the contact was identified. It is user-definable.
- **Field 5**: This is the information from the fifth field of the Telephone Number import, if the contact was identified. It is user-definable.
- **Field 6**: This is the information from the sixth field of the Telephone Number import, if the contact was identified. It is user-definable.

17.6 CallNew

CallNew

This event is fired when a call is first presented at the current extension, be it internal or external, inbound or outbound. It is also fired when an e-mail is routed to the extension.

Parameters:

- **DN**: For external calls, this is the line that the call is on. For internal calls it is the extension called or calling.
- **DNIS**: For inbound direct dialed external calls, this is the DNIS text of the DID number. For other external calls, this is the description of the line that the call is on. For internal calls, the text [Internal] is used.
- **Caller ID** : For inbound external calls, this is the number of the calling party if the information was made available or the string [No Caller ID received!] if no Caller ID was provided. For all other calls, it is the number or extension that the call is connected to.
- **DID** : For direct dialed inbound external calls, this is the direct dial digits entered by the caller. The format of this string is telephone system dependent, but is usually either the pertinent section of the dialed number, or the entire dialed number.
- **Serial Number**: For external calls, this is a unique string identifying the call. The MiCC Office Server uses this text to identify the call.
- **Call Flags**: This bitfield holds various information on the state of the call, as follows:

Value	Description
1	Internal call
2	Outbound call
4	Call held
8	Call answered
16	Contact identified in Telephone Import database
32	Caller ID received

- **Field 2**: This is the information from the second field of the Telephone Number import, if the contact was identified. Usually it is the contact's name or company name.
- **Field 3**: This is the information from the third field of the Telephone Number import, if the contact was identified. It is user-definable.
- **Field 4**: This is the information from the fourth field of the Telephone Number import, if the contact was identified. It is user-definable.
- **Field 5**: This is the information from the fifth field of the Telephone Number import, if the contact was identified. It is user-definable.
- **Field 6**: This is the information from the sixth field of the Telephone Number import, if the contact was identified. It is user-definable.

17.7 CallRemoved

CallRemoved

This event is fired when a call at the current extension is removed, either because the callee or caller dropped the call.

Parameters:

- **DN**: For external calls, this is the line that the call is on. For internal calls it is the extension called or calling.
- **DNIS**: For inbound direct dialed external calls, this is the DNIS text of the DID number. For other external calls, this is the description of the line that the call is on. For internal calls, the text [Internal] is used.
- **Caller ID** : For inbound external calls, this is the number of the calling party if the information was made available or the string [No Caller ID received!] if no Caller ID was provided. For all other calls, it is the number or extension that the call is connected to.
- **DID** : For direct dialed inbound external calls, this is the direct dial digits entered by the caller. The format of this string is telephone system dependent, but is usually either the pertinent section of the dialed number, or the entire dialed number.
- **Serial Number**: For external calls, this is a unique string identifying the call. The MiCC Office Server uses this text to identify the call.
- **Call Flags**: This bitfield holds various information on the state of the call, as follows:

Value	Description
1	Internal call
2	Outbound call
4	Call held
8	Call answered
16	Contact identified in Telephone Import database
32	Caller ID received

- **Field 2**: This is the information from the second field of the Telephone Number import, if the contact was identified. Usually it is the contact's name or company name.
- **Field 3**: This is the information from the third field of the Telephone Number import, if the contact was identified. It is user-definable.
- **Field 4**: This is the information from the fourth field of the Telephone Number import, if the contact was identified. It is user-definable.
- **Field 5**: This is the information from the fifth field of the Telephone Number import, if the contact was identified. It is user-definable.
- **Field 6**: This is the information from the sixth field of the Telephone Number import, if the contact was identified. It is user-definable.

17.8 CallRetrieved

CallRetrieved

This event is fired when a call is retrieved (having previously been on hold) at the current extension.

Parameters:

- **DN**: For external calls, this is the line that the call is on. For internal calls it is the extension called or calling.
- **DNIS**: For inbound direct dialed external calls, this is the DNIS text of the DID number. For other external calls, this is the description of the line that the call is on. For internal calls, the text [Internal] is used.
- **Caller ID** : For inbound external calls, this is the number of the calling party if the information was made available or the string [No Caller ID received!] if no Caller ID was provided. For all other calls, it is the number or extension that the call is connected to.
- **DID** : For direct dialed inbound external calls, this is the direct dial digits entered by the caller. The format of this string is telephone system dependent, but is usually either the pertinent section of the dialed number, or the entire dialed number.
- **Serial Number**: For external calls, this is a unique string identifying the call. The MiCC Office Server uses this text to identify the call.
- **Call Flags**: This bitfield holds various information on the state of the call, as follows:

Value	Description
1	Internal call
2	Outbound call
4	Call held
8	Call answered
16	Contact identified in Telephone Import database
32	Caller ID received

- **Field 2**: This is the information from the second field of the Telephone Number import, if the contact was identified. Usually it is the contact's name or company name.
- **Field 3**: This is the information from the third field of the Telephone Number import, if the contact was identified. It is user-definable.
- **Field 4**: This is the information from the fourth field of the Telephone Number import, if the contact was identified. It is user-definable.
- **Field 5**: This is the information from the fifth field of the Telephone Number import, if the contact was identified. It is user-definable.
- **Field 6**: This is the information from the sixth field of the Telephone Number import, if the contact was identified. It is user-definable.

17.9 DNDStatusChanged

DNDStatusChanged

This event is fired when the do-not-disturb status at the current extension changes.

Parameters:

- **Enable**: This value is 0 if DND is being disabled, and 1 if being enabled.
- **DND Message**: This is the DND message.
- **DND Text**: This is the DND text.

17.10 ExtAccountCodeEntered

ExtAccountCodeEntered

This event is fired when an account code is entered into the handset while on an active call.

Parameters:

- **Account Code**: The account code that was entered against the call.
- **Trunk Line**: The trunk line of the external call that the account code was entered for.

17.11 ExtAgentLogon

ExtAgentLogon

This event is no longer fired, as of 2.5. To track agent log on/off, use the **ExtAgentStatusChanged** event, which has been updated to support this functionality.

17.12 ExtAgentStatusChanged

ExtAgentStatusChanged

This event is fired when an ACD agent's status changes.

Parameters:

- **Agent Status**: A number between 0 and 8 depicting the status that the ACD agent changed to, as follows:

  0 – Logged Out
  1 – Logged In
  2 – Free
  3 – Busy (Call)
  4 – Busy (E-mail)
  5 – Wrapup (Call)
  6 – Busy N/A (DND)
  7 – Wrapup (E-mail)
  8 – Free (E-mail)

- **ACD Agent ID**: The agent ID whose status has changed.

- **Non-Agent Login Count**: This is a number depicting the number of times that the ACD agent has logged into a non-agent ID type hunt group.

- **Agent Login Count**: This is the number of times that the ACD agent has logged into an agent ID type hunt group.

- **Hunt Group**: This is the ACD hunt group that the agent is logging into.

17.13 ExtDigitsToVM

ExtDigitsToVM

Although it appears in Visual Basic, this event is not used.

17.14 ExtDivertStatusChanged

ExtDivertStatusChanged

This event is fired when the divert status at the current extension changes.

Parameters:

- **Forward State**: A number between 0 and 4 representing the forward / divert state being changed to, as follows:

  0 – None
  1 – Immediate
  2 – No Answer
  3 – On Busy
  4 – No Answer / On Busy

- **Device**: The extension device or group being diverted to.

17.15 ExtLostCall

ExtLostCall

This event is fired when a call alerting the current extension is dropped before being answered. The event is fired only if this is the first extension that the call has been presented to.

Parameters:

- **Trunk Line**: The trunk line number that the call was active on.
- **DNIS**: A string containing one of the following values:

  - The DID DNIS description for inbound external DID calls.
  - The trunk line description for external non- DID calls.
  - “Internal” for internal calls.

- **Caller ID** : The received Caller ID or dialed digits for the call. If no Caller ID was received for an inbound external call, then this string contains “[No Caller ID]”.
- **DID** : The DID digits for an inbound DID call, or an empty string for non- DID calls.
- **Serial Number**: The unique serial number for the call. The serial number is an internally generated string that the Contact Center Server assigns to each external call.
- **Field 2**: This contains Field 2 from the matched record in the Contact Center Server Telephone Import database.
- **Field 3**: This contains Field 3 from the matched record in the Contact Center Server Telephone Import database.
- **Field 4**: This contains Field 4 from the matched record in the Contact Center Server Telephone Import database.
- **Field 5**: This contains Field 5 from the matched record in the Contact Center Server Telephone Import database.
- **Field 6**: This contains Field 6 from the matched record in the Contact Center Server Telephone Import database.

17.16 ExtMessageToSupervisor

ExtMessageToSupervisor

Although it appears in Visual Basic, this event is not used.

17.17 Idle

Idle

This event is fired when the handset is replaced at an extension.

Parameters:

None.

Mitel
Powering connections

mitel.com