

CallHelix 2.1.2

**by
Autograph Systems**

CallHelix 2.1.2

by
Autograph Systems

Legal

CallHelix License

Please read this license carefully before using the software; by using the software, you are agreeing to be bound by the terms of this license. If you do not agree to these terms, promptly destroy all copies of the software in your possession.

This manual, the CallHelix scripting addition, the Helix Scripting application, and the sample code included with this installation (hereafter jointly referred to as "CallHelix") are copyrighted, with all rights reserved. This copyright is owned by Autograph Systems, and is protected by United States copyright laws and international treaty provisions. Therefore, you must treat the software like any other copyrighted material (e.g., a book or musical recording). CallHelix may not be copied, except as otherwise provided in your software license or as expressly permitted in writing by Autograph Systems.

Use of CallHelix and its documentation are governed by the terms set forth in this license. Such use is at your sole risk. CallHelix is provided "as is" and without warranty of any kind and Autograph Systems expressly disclaims all warranties and/or conditions, express or implied, including, but not limited to the implied warranties and/or conditions of merchantability and fitness for a particular purpose.

Autograph Systems does not warrant that the functions obtained in CallHelix will meet your requirements, or that the operation of CallHelix will be uninterrupted or error free, or that defects in CallHelix will be corrected. Furthermore, Autograph Systems does not warrant or make any representation regarding the use or the results of the use of CallHelix or related documentation in terms of their correctness, accuracy, reliability, or otherwise, without prejudice to the generality of the foregoing. No oral or written information or advice given by Autograph Systems or any of its authorized representatives shall create a warranty or in any way increase the scope of this warranty.

Should CallHelix prove defective, you (and neither Autograph Systems nor any of its authorized representatives) assume the entire cost of all necessary servicing, repair, or correction. Some jurisdictions do not allow the exclusion of implied warranties, so the above exclusion may not apply to you. The terms of this disclaimer and the limited warranty do not affect or prejudice the statutory rights of a consumer acquiring this product otherwise than in the course of a business, neither do they limit or exclude any liability for death or personal injury caused by negligence. Under no circumstances, including negligence, shall Autograph Systems or any of its authorized representatives be liable for any incidental, special or consequential damages that result from the use or inability to use CallHelix, even if Autograph Systems or any of its authorized representatives has been advised of the possibility of such damages. Some jurisdictions do not allow the limitation or exclusion of liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. In no event shall Autograph Systems' total liability to you for all damages, losses, and causes of action (whether in contract, tort [including negligence], or otherwise) exceed the amount paid by you for CallHelix.

CallHelix (full package) is ©1999-2008 by Autograph Systems, 72 Sherwood Street, Mansfield, PA 16933. All rights reserved.

CallHelix (scripting addition) is based on code ©1997 by Communications Unlimited, Le Domaine St. Pierre, F-84850 Travaillan, France. All rights reserved.

Helix, Helix RADE, Helix Server, Helix Client, and Helix Engine are trademarks of QSA ToolWorks, LLC.

Apple, the Apple logo, AppleShare, AppleScript, AppleTalk, LocalTalk, Macintosh, Mac OS, and Script Editor are trademarks of Apple Inc.

All other trademarks and service marks are the property of their respective owners.

Technical

About CallHelix:

Concept, Manual: Matthew Strange, Autograph Systems

Programming: Ryan Wilcox, Wilcox Development Solutions

Original Concept: Fred Stevenson, Communications Unlimited

Sales, Licensing, and Technical Support:

Autograph Systems <matt@autographsystems.com>

Version info:

This manual corresponds to CallHelix version 2.1.2

Release Date: October 13, 2008

General CallHelix information is found at:

<<http://scripting.autographsystems.com/callhelix/>>

Current version information is found at:

<<http://scripting.autographsystems.com/callhelix/currentversion.html>>

CallHelix purchase information is found at:

<<http://scripting.autographsystems.com/callhelix/purchase.html>>

CallHelix Demo version is found at:

<<http://scripting.autographsystems.com/callhelix/demo.html>>

Table of Contents

Introduction	1
What is 'CallHelix'?	
What is 'Helix Scripting'?	
What is Helix?	
What is AppleScript?	
How AppleScript, CallHelix, and Helix work together	
What is new in version 2.1.2?	
Chapter 1: Getting Started	3
Package contents	
Installing CallHelix	
What else do I need to know?	
Conventions used in this manual	
Chapter 2: AppleScript Overview	5
AppleScript conventions	
Learning to write AppleScripts	
Chapter 3: Quick Start	8
Using Helix Scripting	
Adding records	
Retrieving records	
Notes on retrieving records	
Chapter 4: CallHelix Basics	11
Foundational introduction	
Sending requests to Helix	
Handling replies from Helix	
Handling errors	
Additional notes	
Chapter 5: Introduction to Complex Processes	16
Chapter 6: Storing Records	17
Complex storage: the process	
Notes on storing records	
Putting Store Records to work	
Chapter 7: Retrieving Records	21
Complex retrieval: the process	
Notes on retrieving records	
Record retrieval options	
Get View Data as ... additional options	
Notes on retrieving records	
Putting Get View Data As ... to work	
Chapter 8: Deleting Records	27
Delete records: the process	
Notes on deleting records	
Putting Delete Records to work	
Chapter 9: Check View State	30
Putting Check View State to work	
Appendix A: Command Reference	31
Appendix B: Helix Apple Event Error Codes	39
Helix specific errors	
CallHelix specific errors	
Appendix C: Helix HAEC Resources	40
Appendix D: CallHelix Demo Version	40
Appendix E: Targeting a Remote Computer	41
Targeting Helix on a remote machine	
Results of various target statements	
Appendix F: Version History	44
Quick Reference Card	45

Introduction

What is 'CallHelix'?

CallHelix is an AppleScript Scripting Addition (aka **OSAX**) that enables you to manipulate data within a Helix database using the AppleScript language. Records in a Helix database can be added, retrieved, and deleted through the users and views that are already designed in a collection. No special programming within Helix is required, but programming with CallHelix in mind can allow you to create some very powerful solutions that are not otherwise possible.

What is 'Helix Scripting'?

Helix Scripting is a faceless background application (FBA) that does everything the CallHelix scripting addition does. The advantages of using an application instead of a scripting addition are subtle, but significant. If you are just beginning to use CallHelix, we recommend that you install Helix Scripting and ignore the CallHelix OSAX.

In this manual, "CallHelix" refers to both the CallHelix OSAX and the Helix Scripting FBA.

What is Helix?

Helix is a relational database program for the Mac OS. In this manual, Helix refers to the Helix RADE, Helix Server, and Helix Engine database products which are trademarks of QSA ToolWorks, LLC. CallHelix also works with the older "Helix Express" products (versions 3.7.1 and higher) but those products are no longer officially supported.

External access to a Helix collection is provided via the Apple Event (AE) protocol. Via Apple Events an external application can add, retrieve and delete records in a Helix collection. It is important to note that Helix does not send Apple Events; it only receives them. As such, external access is controlled totally by the external application. CallHelix is the bridge between the Apple Events in Helix and AppleScript.

Helix 5.0 (released September, 2000) corrected a number of bugs to the original Helix implementation of AE. Calls that failed in earlier versions work correctly in Helix 5.0 and later. Most significantly, Helix 5.0 correctly implements the "post on retrieve" specification. With Helix 5.0 and later, posts attached to the "On Export" column are triggered when a record is retrieved.

Subsequent Helix releases have continued the trend of fixing bugs and adding new features. Check the Helix release notes for the latest information.

Helix 6.0 (released December 19, 2005) featured an OS X native Helix Server. No new Apple Event features were added to Helix 6.0, but this release makes it possible to take full advantage of the OS X native components of CallHelix.

What is AppleScript?

[From [www.apple.com/applescript/...](http://www.apple.com/applescript/)]

AppleScript is an English-like language used to create script files that control the actions of the computer and the applications that run on it. Much more than just a macro-language, which simply repeats your recorded actions, AppleScript scripts can "think." Scripts can make decisions based on user-interaction or by parsing and analyzing data, documents or situations. AppleScript scripts can automate much of what we do, make your time spent of the computer more productive, less stressful, and save time and money.

How AppleScript works

When you tell your computer to do a task (for example, by choosing a menu command), the operating system and applications on your computer talk to each other using a messaging tool called Apple Events. Apple Events transfer information, commands, and requests between applications, networks, and the Mac OS – much like we use phone calls, pagers, or email to communicate with each other.

When you send an AppleScript message, the operating system converts the script's instructions into Apple Events messages, then sends the events to the indicated applications where the instructions are executed.

To learn more about AppleScript and how to use it, choose Mac Help from the Finder's Help

menu, then search for AppleScript. There you can find an overview of AppleScript features, as well as background information and instructions on how to use AppleScript.

How AppleScript, CallHelix, and Helix work together

CallHelix takes Helix's Apple Event commands (normally accessible only to programmers using high level languages such as C/C++) and makes them available via AppleScript. The primary function of CallHelix is to act as a simple interpreter, converting the AppleScript commands that you understand into Apple Events that Helix understands. CallHelix 2.1 extends this concept, adding special timesaving commands that result in significant improvements in code size, speed, and readability.

What is new in version 2.1.2?

Version 2.1.2 addresses Mac OS X 10.5 compatibility issues.

Changes ...

1. Works around an issue that results in blank records being created in Helix. This happens because Leopard silently changes the input to Unicode data, and Helix expects ASCII data. CallHelix now explicitly converts all data to ASCII before sending it to Helix.
2. Fixes a bug in Helix Scripting that caused it to be referred to as "Helix Scripting" in some places and "Helix Scripting.app" in others. The result is that scripts (written using one name or the other) could fail if they were unable to find Helix Scripting with the expected name.

What is new in version 2.1.1?

Version 2.1.1 is a bug fix release that addresses bugs in CallHelix 2.1.

Bug fixes ...

1. A bug in Selector 143 (get partial view data) caused it to ignore the Subform Tabbing parameter.
2. Copyright notice text corrected.

What is new in version 2.1?

Version 2.1 adds a new feature that allows the target parameter to directly target a Helix application running on a remote machine. It is no longer necessary to install CallHelix on the same machine as the Helix application.

New features ...

1. Enhanced target parameter allows direct connection to remote Helix applications.

What is new in version 2.0?

Version 2.0 is a significant upgrade to CallHelix. It features many improvements over earlier versions of CallHelix. If you are already familiar with the CallHelix syntax, you can simply install CallHelix 2.0 and see immediate results from faster script execution. As you learn to utilize the new commands, you can rewrite your scripts and reduce multiple lines to a single call, making both script writing and execution significantly faster.

New features ...

1. PowerPC native code: scripts run 30% faster (or more).
2. OS X native code: write and run scripts in OS X.
3. New faceless background application: easier communications.
4. New commands: significantly streamlines script writing.
5. New syntax: makes code more readable and faster to write.
6. New retrieval capabilities: retrieve delimited text or lists of field data.
7. Target Helix application: address multiple copies of Helix on one machine.
8. CallHelixVersion command: easy test for code compatibility.
9. Integrated Error Code handler: meaningful text messages.
10. Demo version: try before you buy.
11. Full Helix 5.x (and later) compatibility: take full advantage of Helix Apple Events.
12. Better conformity with Helix's Apple Event specifications.

Chapter 1: Getting Started

Package contents

CallHelix is available in three variations:

1. **Helix Scripting** – the faceless background application (FBA) that can be installed on any version of Mac OS.
2. **CallHelix.osax** – the scripting addition for installation on machines running OS X.
3. **CallHelix** – the original scripting addition, for machines running OS 8 or 9.

Note: despite its name, ‘Helix Scripting’ is really just another CallHelix variant. This manual uses the generic term ‘CallHelix’ to refer to all three products.

In addition to CallHelix, the installation includes:

1. **CallHelix Tutorial** – a simple collection that works with the code in this manual. Also includes the scripts for the longer ones, so you don’t need to type them in.
2. **Extras** – script samples that illustrate various CallHelix techniques and applications. These scripts are unsupported, but are provided to help you learn to use CallHelix effectively and to give you ideas for your own application. More samples can be found at the CallHelix web site: <http://scripting.autographsystems.com/callhelix/>

Installing CallHelix

If you are just beginning to use CallHelix, it is recommended that you install the Helix Scripting FBA, not the CallHelix OSAX.

Scripts running natively in OS X are able to communicate with Helix databases running in Classic mode on that machine, so even if you are working with a Classic Mode version of Helix, you should install CallHelix.osax (not CallHelix) on an OS X machine.

Determining which CallHelix variation to install

Before you install, you must first determine which variation of CallHelix best fits your needs. Each has its own strengths and weaknesses. You only need to install one of the three CallHelix variations on a machine to enable full functionality. (More than one may be installed if desired.)

Helix Scripting: The faceless background application (FBA) is best suited for writing scripts that access Helix databases running on the same machine, on another machine on the local network, or even on remote machines accessible via TCP/IP. The Helix Scripting FBA can be installed on Macs running any version of Mac OS 8.0 or later. If you are new to CallHelix and you intend to take full advantage of CallHelix’s capabilities, this is the CallHelix variation to install. The main disadvantage is that you must address it explicitly, as opposed to simply issuing commands and letting the OS route them. (This may actually be considered an advantage, as it makes a sharp distinction between system functions and CallHelix functions, eliminating potential ambiguities.)

CallHelix.osax: This is the OS X native version of the CallHelix scripting addition. If you are upgrading from an earlier version of CallHelix or are working with scripts written for CallHelix scripting addition, this is the best to install.

The disadvantages of installing a scripting addition are that it loads every time the Mac boots up, taking up memory regardless of whether you are using it or not. In addition, scripting additions works best when the scripts run on the same machine as the Helix application itself.

CallHelix: This (the original) scripting addition runs only under OS 8 and 9. (It can also be installed in the Scripting Additions folder of a System Folder of a machine running OS X, enabling Classic mode access to the scripting addition.) However, if you are running OS X, installing the OS X native version (*CallHelix.osax*) results in much better performance.

Installing CallHelix

Helix Scripting: Because *Helix Scripting* is an application, it can be installed anywhere. However, if you install it in the standard location for scripting additions (the Library:ScriptingAdditions directory in OS X or the Scripting Additions folder in OS 8–9) the system should be able to locate and automatically launch it when it is needed. To install Helix Scripting in the Scripting Additions folder, use the instructions for installing CallHelix.osax.

CallHelix.osax: Installing under OS X requires that you create a folder specifically for scripting additions if one does not already exist. Check your Library folder (either the Library folder at the root level of the hard disk or the Library folder in your home directory) for a folder named “ScriptingAdditions” (no spaces). If one does not exist, create a new folder, name it, and copy the CallHelix.osax into that folder. Note: if you install a scripting addition in the Library of the root directory, it is available to all users of this system. If you install it in

your home Library folder, it is only available when the system is logged in under your name.

CallHelix: To install CallHelix, just drop the file named *CallHelix* onto your (Classic) System Folder. CallHelix is automatically routed to the Scripting Additions folder. If you are upgrading from CallHelix 1.x, you should be prompted to replace your old version. If you are not prompted to replace an older version, you should open the Scripting Additions folder and delete the old version so that it is not inadvertently called by your scripts. After installing CallHelix, restart the Mac. Existing scripts run exactly as they did before, but faster.

Licensing reminder

Under the terms of this license, you may install CallHelix on up to two computers.

To use either the CallHelix or CallHelix.osax scripting addition, a copy must be installed on the computer that is running the Helix database. If you are using one of the scripting additions to work with a remote Helix database – typically a Helix Server – one of your licensed copies of CallHelix must be installed on that machine.

Since the Helix Scripting FBA does not need to be installed on the machine where the Helix database is running, you are free to install it on any two computers you wish.

If you intend to use CallHelix on a regular basis on multiple Macintoshes, please purchase the appropriate multiple CPU or site license.

What else do I need to know?

Typically you use Apple's Script Editor to write your AppleScripts, but any OSA compliant scripting environment can be used. Scripts can be written to run on the same computer the Helix collection is running on, or from a different machine on your network.

Note that under Mac OS 9.0 and later, Program Linking can also function via a TCP/IP (Internet) connection, making it possible to write AppleScripts that can communicate with a Helix database anywhere in the world. The possibilities (both good and bad) this opens up are staggering. Before you head into this uncharted territory, make sure you have covered all of the possible security holes that are opened when you make your database available to the world.

If you plan to access CallHelix over your network, Appendix E: Targeting a Remote Computer discusses issues related to remote AppleScript access.

Pictures and documents can not be passed via Apple Events. If you want to work with these data types, you should use Helix's document handling features and store the files externally. (Beginning with Helix 5.2, retrieving a document returns its path.)

A reminder: in this manual, the term "CallHelix" applies to both CallHelix and Helix Scripting.

Important changes in Helix

CallHelix communicates with Helix through Helix's Apple event functionality. The Apple event capabilities of Helix have been improved and enhanced in recent releases. In addition, Helix 5.1 introduced a number of terminology changes, attempting to bring Helix terminology into line with industry standards. For example, the terms ***Dump*** and ***Load*** have been changed to ***Export*** and ***Import***.

As of this writing, the current version of Helix is 6.0. This manual attempts to point out differences in the Helix Apple event implementation from version to version, but is primarily written with the Helix 6.0 specification in mind.

Conventions used in this manual

Important words (usually defined elsewhere in the manual) are set in ***bold italic type***.

Important notes are set in bold type.

AppleScript code snippets are set as follows:

display dialog "AppleScript code is written in ITC Officina Sans in the CallHelix manual"

Special characters, (such as ***tab*** and ***return***) are difficult to show in a manual. These symbols are used to illustrate those characters:

→ Tab – This symbol is also used to generically refer to field delimiters.

↵ Return – This symbol is also used to generically refer to record delimiters.

Chapter 2: AppleScript Overview

AppleScript conventions

AppleScript's syntax is similar to other programming languages, but since Helix is a visual, icon based language, you may not be familiar with the conventions used by AppleScript. This manual is not an AppleScript tutorial, but here is a quick overview of the common terminology used when communicating with CallHelix.

Reserved words

Words that have special meaning and can't be used for other purposes. Commands, Operators, and Constants are all considered reserved words. For example, `set` is a command (as in `set myNumber to 7`) so you can't use `set` as a variable name. `space` is a constant (as in `if myData contains space`) and can't be used for any other purpose.

Commands (Keywords)

Words that instruct AppleScript to perform an action. Examples: `tell`, `set`, `CallHelix`. Most AppleScript lines start with a command, as in `tell application "Helix RADE" to quit` or `set aVariable to aValue`.

Operators

Words or symbols that instruct AppleScript to perform an action on two values, combining the two into a new value. Examples: `+`, `or`, `contains`, `&`

Constants

Words that have a predefined meaning. Constants are part of the AppleScript language and protected from being redefined. Attempting to assign a value to a constant results in an error. Examples: `return`, `space`, `pi`

Data types

AppleScript supports a wide variety of data types, including the field (text), number (real), and flag (boolean) types used in Helix. Other AppleScript data types commonly used when working with Helix are ...

Lists

A grouping of strings and/or variables, enclosed in curly brackets `{ }` and separated by commas. Example: `{"a string", aVariable, Text Item 3 of myList}`

Note: CallHelix makes extensive use of lists; parameters are sent to Helix in lists and many replies come back from Helix in lists. Understanding how lists function (and how to coerce lists to strings (and strings to lists)) is vital to working with CallHelix effectively.

Records

Similar to a list, a record is a grouping of strings and/or variables, enclosed in curly brackets `{ }` and separated by commas. However, records carry an additional facet in that each item in the list is also given a specific name, making it possible to locate a piece of data in a record without knowing its ordinal position in the list. Example: `{theString: "a string", value: aVariable, keyword: Text Item 3 of myList}`

Note: CallHelix makes extensive use of records when returning data from Helix. For example, when setting up a process to retrieve data from Helix, CallHelix first returns the number of records on the view along with the field and record delimiters for that view. This data is returned in a record, as in `{record count: 25, field delimiter: tab, record delimiter: return}` and each item in the record is handled by addressing it by name, as in: `set theCount to (record count of the result)`. Understanding how records function is vital to working with CallHelix.

Lists vs. Records

A list's data is defined by the order in which items appear. Conversely, a record's data is defined by the names assigned to each item. Working with records and items is a familiar concept to Helix users, but keep in mind that during interaction between Helix and AppleScript, no attempt is made to correlate Helix field names to AppleScript item names. Helix's Apple Event interaction is ordered strictly by the layout of the view that is being addressed. In CallHelix, records are used to differentiate between the *types* of data that are returned by Helix, but they have no usefulness when working with the *actual record data* returned by Helix. For instance, if you know there is a Helix field named "Last Name" on a view, and you use CallHelix to retrieve data from that view, but you can not locate that field by writing `get last name of the result`.

Working with data

Data can be assigned in various ways in AppleScript. Common methods ...

Literals

A fixed value, typed into the script. Visually, a literal just looks like the value it represents. Text (string) literals are enclosed in double quotes, as in "Nancy". Numeric literals are entered as numbers, as in 7.25. Boolean literals are true and false.

Variables

Temporary/redefinable data holders. Visually, a variable looks like a single unquoted word (or multiple words not separated by spaces). AppleScript supports both local and global variables. Examples: MyVariable, last_name, tempnumber.

Properties

Persistent variables. Typically defined at the beginning of a script, properties look and feel like global variables. Their chief advantage is that the stored value is retained even after the script is closed. Example: property myCollectionName: "Bank Records"

Coercion

Converting a piece of data from one type to another. Also known as **typecasting**. In Helix the Number, Text, Fixed Point, Date, Time, and Picture tiles are used to explicitly coerce data from one type to another. In AppleScript, coercion can be either implicit or explicit. If you concatenate a piece of text and a number, AppleScript implicitly attempts to coerce the number to a string. The statement `set theString to "I am " & 34 & "."` coerces the number 34 into a string, resulting in "I am 34." Usually this implicit coercion (the first data type encountered determines the resulting data type) works fine, but you also have the option of explicitly coercing the data. The statement `set theString to "I am " & (34 as text) & "."` or `set theString to ("I am " & 34 & ".") as text` gives the same result as the former statement, but removes the possibility of a misunderstanding.

Handlers

Programmer-defined subroutines that *handle* predefined tasks. Also referred to as **functions** or **procedures**, handlers make code easier to manage. Handlers are defined by the `on` or `to` command and can both receive and return variable data. For example, an `on AddRecord(myData)` handler can be defined to handle all record adding tasks the script has to accomplish. By defining the handler once and passing in the appropriate data, one routine can be used repeatedly, making code more compact and easier to maintain.

Parameters

Values that are used to carry out a command. In `set myVariable to "hello"` both `myVariable` (a variable) and `"hello"` (a string literal) are **parameters** that the `set` command has been given to work with.

A **parameter list** is simply a set of parameters in list form, as in `set paramList to {"MyCollection", "MyUser", "MyPass", "MyRelation", "MyView"}`

When you call a handler, you can pass parameters to it, as in `AddRecord(myData)`, where `AddRecord()` is a handler and `myData` is a parameter.

When you communicate with Helix, you usually start by passing a **parameter list** containing the "basic 5" elements: collection name, user name, password, relation name, and view name. This can be done as a parameter list: `AddRecord(paramList, myData)` or as distinct parameters: `AddRecord("MyCollection", "MyUser", "MyPass", "MyRelation", "MyView", MyData)`.

Comments

Explanatory notes that are not executed by AppleScript. Two types of comments exist in AppleScript. Single line comments are preceded by two dashes:

`display dialog "there is a comment on this line" -- this is a single line comment; it won't be treated like a command.`

Longer, multiple line comment sections are bracketed by `(*` and `*)` :

`(* This is a very long description of the multiple line comment command.`

`By enclosing this text in the symbols shown here I can write multiple lines of text
and it is all ignored by AppleScript. *)`

Learning to write AppleScripts

You can find detailed AppleScript information, tutorials, scripts, and links to other resources on the AppleScript web site: <http://www.apple.com/applescript/>. Many additional resources can be found at <http://www.apple.com/applescript/resources/>.

Update Note: CallHelix 2 adds parameters for specifying a specific copy of Helix, and for controlling the format of the returned data. If you are upgrading from CallHelix 1.x, be sure to familiarize yourself with the time savings available through the new parameters.

If you are just starting to write AppleScripts, Apple's free Script Editor is sufficient, but as your scripts become more complex, the limitations of Script Editor (think of it as the *Simple Text* of script editors) become apparent. If you plan on spending a significant amount of time working with AppleScript, you should invest in a professional script editor. We recommend Script Debugger, from Late Night Software <<http://www.latenightsw.com/>>. A complete list of alternative script editors can be found at <<http://www.apple.com/applescript/>>.

A number of excellent books have been written about AppleScript. If you are new to AppleScript, check amazon.com or your local bookstore for the latest introductory guides. Once you've gotten a handle on the basics, Matt Neuburg's *AppleScript: the Definitive Guide* is highly recommended.

Autograph Systems is available to help you create CallHelix-based solutions. Support is available on an hourly basis. Contact us for more information.

Commenting your code

As you learn to write AppleScripts, you should force yourself to comment your code thoroughly. Remembering the specifics of a script's functions is often difficult – even more so as time passes.

A useful technique is to include the date a line or routine was written. As you learn, you may revisit some of your early scripts and run across routines that aren't immediately discernible. Dating them can help you recall your level of expertise and determine whether or not a routine should be updated to incorporate your current skill level.

Chapter 3: Quick Start

CallHelix provides single line commands that enable you to quickly begin to tap into the power of using AppleScript to communicate with Helix. This chapter presents those simple commands and introduces concepts that are built upon in later chapters.

Using Helix Scripting

When writing AppleScripts, you must precede your commands with a tell statement directing the commands to the correct application: for Helix that is Helix Scripting:

```
tell application "Helix Scripting"  
    CallHelix ( ... the rest of the line goes here )  
end tell
```

Note that if you are using one of the scripting additions instead, you must leave the tell and end tell lines out. Although these extra lines make your code a little larger, learning to direct your command explicitly to an application (Helix Scripting in this case) to with a tell statement pays off when you begin to write more complex scripts.

Some of the examples below omit the tell wrapper, mostly to save space. Remember: if you are using Helix Scripting, they are always required, and if you are using one of the scripting additions, they must be left out.

Adding records

Adding a record to a Helix database

Adding records to a database requires that you know the name of the collection, the relation the record is to be stored in, the view through which you want to add the record, and a user (and the user's password) that has that view on their menu. Given those parameters, all that is left is to pass the parameters to Helix in the proper order and supply the data to be stored.

Any time you access a view in Helix, you must supply the "Basic 5" parameters, in the correct order (collection name, user name, user password, relation name, view name). When using the store one record command, the data is sent as the sixth parameter.

Sending data into Helix from CallHelix is very similar to using Helix's Import command. In a text file, data for each field is separated by a tab character. Likewise, to send multiple fields of data as a single (6th) parameter, the data must be *concatenated* into a single text string with tabs embedded as *delimiters* for the fields.

Concatenating data in AppleScript is done with the *&* operator. To format data that is intended to be stored in consecutive fields on a view, you concatenate the data ...

```
("sent from" & tab & "CallHelix") as string
```

... creating a single text string ...

```
"sent from→CallHelix"
```

... which is treated as a single parameter, the required format for **store one record**:

```
tell application "Helix Scripting"  
    CallHelix {"MyCollection", "MyUser", "MyPass", "MyRelation", "MyView", ("sent from" & tab & "CallHelix") as string} action store one record  
end tell
```

This single line of AppleScript creates a new record in the relation "MyRelation" with the words "sent from" in the first field found on the view "MyView", and "CallHelix" in the second.

Although most of the examples in this section do not use variables, the concept should be introduced here because without it, handling data containing many fields can quickly become unwieldy. Using a variable to hold the data to be sent to Helix, we can rewrite the previous line as follows:

```
tell application "Helix Scripting"  
    set theWholeRecord to ("sent from" & tab & "CallHelix") as string  
    CallHelix {"MyCollection", "MyUser", "MyPass", "MyRelation", "MyView", theWholeRecord} action store one record  
end tell
```

By using a variable (theWholeRecord) to hold the data to be stored, the second line (where the data is actually sent to Helix) is kept short and manageable, and is much easier to read.

Notes on adding records

The view controls the field data that can be added, the field delimiter, and the order that data must be sent in. The data must be formatted as plain text, with no start or stop char-

acters. Pictures and documents may not be sent, and the field delimiter (tab in this example) must correspond to the field delimiter set in the view's import/export options dialog.

Store One Record respects all field validations on the entry view. If data sent into Helix generates an error – whether a validation failure or a data type error – Helix returns error 1000 (“An error was encountered trying to store data on an entry view”) just as it would if you attempted to import the data from a text file. The **On Error** settings in the view's import/export options dialog are also honored.

Store One Record handles one record per call. Record delimiters are passed through as field data. Therefore, if the view's record delimiter is the return character, sending ...

```
tell application "Helix Scripting"
  CallHelix {"MyCollection", "MyUser", "MyPass", "MyRelation", "MyView", "my" & return & "data"} action store one record
end tell
```

... results in a single record with the text “my↵data” stored in the view's first field, with the words ‘my’ and ‘data’ on two separate lines, just as if you typed the data on the view.

Store One Record triggers posts that are set in the view's **On Entry** column.

Store One Record only works with entry views. If you try to store data in a list view, Helix returns error 200 (Not an entry view).

Store One Record does not support passwords in Helix 4.x. This bug was fixed in Helix 5.0.

For storing a few records, **Store One Record** is sufficient, but if you have many records to store, the **store records** command (described in **Chapter 5: Introduction to Complex Processes** on page 16) should be used.

Retrieving records

CallHelix has two powerful **Retrieve Records** commands that reduce the entire process to a single command.

Retrieving records from a Helix database

Retrieving records from a database (analogous to using Helix's **Export All** command) requires that you know the **Basic 5** parameters: the name of the collection, the relation the record is to be stored in, the view you want to retrieve records from, and a user (and the user's password) that has that view on their menu. Given those parameters, all that remains is to pass the parameters to Helix (in the proper order) and to specify whether you want each record returned as a delimited text string or divided into lists, with each field treated as a distinct piece of data.

Retrieve each record as a single entity (delimited data)

For basic tasks, such as moving data from one relation (or collection) to another, or for writing the data out in a text file, the retrieve records as string command, which retrieves each record as a text string—complete with its Helix field and record delimiters—is the most efficient option. The field data for each record is returned as a string, with the field delimiter between each field and the record delimiter at the end. This is the same format as data retrieved via the **Export Records** and **Copy Records** commands in Helix.

To retrieve data on a view, all that is needed is to pass the **Basic 5** parameters:

```
tell application "Helix Scripting"
  set theHelixData to CallHelix {"MyCollection", "MyUser", "MyPass", "MyRelation", "MyView"} action retrieve records as string
end tell
```

CallHelix returns all of the data available through that view. The data is returned as a list, and each element of the list is an AppleScript record containing an item named helix record. Using the sample collection in the CallHelix package, this is the result:

```
{ {helix record:"Record 1→Field 2→1↵"}, {helix record:"Record 2→Field 2 in Record 2→2↵"} }
```

Find out how many records were returned with the line:

```
set recordCount to (count items in theHelixData)
```

Access the data in each record by using a repeat loop:

```
tell application "Helix Scripting"
  repeat with each in theHelixData
    set thisRecord to (helix record of each)
  end repeat
end tell
```

Retrieve each record as multiple entities (list data)

Although there are times when working with each record as a single entity is sufficient, it is

more common to want to access the fields within those records. For these tasks, the `retrieve records as list` command is provided. This command retrieves each record as a list of data, each field being separated into a distinct list element.

To retrieve data on a view, simply pass the five required parameters:

```
tell application "Helix Scripting"
    set theHelixData to CallHelix {"MyCollection", "MyUser", "MyPass", "MyRelation", "MyView"} action retrieve records as list
end tell
```

`CallHelix` returns all of the data available through that view, storing it in the variable `theHelixData`. `theHelixData` contains a list of AppleScript records, each one containing a single item named `helix record`. The contents of `helix record` is a list of AppleScript items, each field in the Helix record placed in consecutive list items. Using the sample collection supplied with `CallHelix`, this is the result:

```
{ { helix record: { "Record 1", "Field 2", "1" } }, { helix record: { "Record 2", "Field 2 in Record 2", "2" } } }
```

Find out how many records were returned:

```
set recordCount to (count items in theHelixData)
```

Access the data in each record by using a repeat loop:

```
tell application "Helix Scripting"
    repeat with each in theHelixData
        set thisRecord to (helix record of each)
    end repeat
end tell
```

Each time this loop repeats, the variable `thisRecord` contains a list of items, with each field from the view stored in consecutive list items. You access the first field by writing ...

```
item 1 of thisRecord
```

... and so on. In this example, the third field on the view is a number, so you can total the contents of that field with the following code:

```
tell application "Helix Scripting"
    set theHelixData to CallHelix {"MyCollection", "MyUser", "MyPass", "MyRelation", "MyView"} action retrieve records as list
    set theTotal to 0 -- it is important to initialize AppleScript variables before attempting to work with them inside a loop
    repeat with each in theHelixData
        set thisRecord to (helix record of each)
        set theTotal to (theTotal + (item 3 of thisRecord))
    end
    display dialog ("Total of field 3 in the records is: " & theTotal) as string
end tell
```

Notes on retrieving records

The Helix view determines the **fields** that are retrieved, the **query** that restricts the records that are retrieved, the **sort order** of the records retrieved, and the field and record **delimiters**.

When `CallHelix` returns records, it returns data as a list of AppleScript records. The Helix data is found in the `helix record` item of each record.

Retrieve records works with both entry and list views.

Retrieve records triggers posts that are set in the view's **On Export** column.

Retrieve records respects record locking if posts are triggered. Unlike Helix's built-in exporting, which partially completes the process and then stops if a locked record is encountered, `CallHelix` retrieval checks every record *before* starting to retrieve data. If a locked record is found, Helix returns error 270 ("A record is write locked") and no data is retrieved.

Important: The `retrieve records` commands return every record that is available through the selected view. If there are many records in the relation, the amount of data returned can overwhelm AppleScript, resulting in sluggish performance. If you are working with views that can return many records, see [Chapter 7: Retrieving Records](#) on page 21 for efficient methods of working with large data sets.

Concluding remarks

This Quick Start guide shows you how to use the `CallHelix` commands that provide access to Helix data through simple, one line commands. Although these commands work quite well with small data sets, working with larger collections often requires a more efficient, albeit more complex approach. The following chapters cover the more complex methods intended for of working efficiently with large amounts of data in Helix.

Chapter 4: CallHelix Basics

Foundational introduction

Communicating with Helix

Access to Helix data is accomplished through users and views within the collection. In many ways accessing data via CallHelix is similar to the traditional Import and Export commands available in Helix. A Helix application (RADE, Engine, or Server) must be running, the collection must be open, and it must be in User Mode. (AppleScript can also be used to open a collection if it isn't already running.) Access to data in a collection initially requires five pieces of information: the "basic 5" parameters:

1. Collection Name
2. User Name (The target view must be present in a User Mode menu)
3. User Password
4. Relation Name
5. View Name (The Custom view name, as seen on the User Mode menu)

In order to access Helix data, you must know the internal structure of the database. If you do not know the names of the users, the views that are on their menus, and the relations those views are in, you are not able to communicate with the collection. By looking at a user's menus, it is relatively easy to determine which views they have access to, but not the relations in which those views appear. If you have access to Design Mode, **Export Collection Details ...** (available when the collection window is the active window) can output a text file that you can use to supply this missing information.

Communicating with your collection

Helix supports Apple Events that allow you to manipulate the data in a collection. (There are no Apple Events to manipulate the *structure* of a collection.) All external access products that communicate with Helix use these same events, and no others. Helix supports four types of interaction:

1. Add records
2. Retrieve records
3. Delete records
4. Check view type (entry or list)

The CallHelix extended command set

In addition to providing direct access to Helix's built-in Apple Events, CallHelix includes extra commands that greatly simplify communications with Helix. Where Helix's built-in Apple Events require that you use 4 distinct calls, along with extra code to make sure the view is ready, and requires that your script retrieve the records in 10 record increments, CallHelix includes an extended command set that enables AppleScript to retrieve records from Helix with a single line of code. The extended command set makes working with Helix significantly easier *as well as* faster, since much of the "heavy lifting" is done by CallHelix itself.

CallHelix and Posting

Posts that are attached to the views you access are triggered as follows:

- Add record: post on entry
- Retrieve record: post on export (Helix 5.0 and later)
- Delete record: post on entry

Additional notes

Helix itself supports a few commands that are accessible directly via AppleScript. A command to determine the version of Helix that is running was added in Helix 5.2, and one to execute a menu command was added in Helix 5.3. See the Helix documentation for information on using those commands.

Helix cannot directly send Apple Events; it can only receive them. Therefore all external access is controlled by your script.

Helix also includes implementations of Apple's 'core events' Open and Quit. See Helix's documentation for information regarding those events.

Sending requests to Helix

How CallHelix communicates with Helix: the request format

In CallHelix, requests to access a collection are sent as command lines made up of 4 (or 5) elements:

1. The **keyword**. Simply the word 'CallHelix' to tell AppleScript to pass this command to Helix.
2. The **parameter list**. A list of values that tell Helix where (or how) to apply the command.
3. The **action verb**. Simply the word 'action' to tell AppleScript that an action is coming next.
4. The **command**. A short phrase that tells Helix what action to perform.
5. **Optional parameters**. For commands that support them, optional parameters allow you to fine tune the data that Helix returns. Optional parameters are typically appended to the line using AppleScript's **with** clause.

The basic construction of a line looks like:

```
CallHelix {parameter list} action command [with optional parameters]
```

For example, to get all of the records on a view (similar to using 'Export All' in Helix):

```
tell application "Helix Scripting"
```

```
    CallHelix {MyCollection, MyUserName, MyPassword, MyRelation, MyView} action retrieve records as string with record IDs
end tell
```

About the parameter list

The parameter list contains one or more pieces of information that Helix needs to carry out the command. For example, to send a single record to Helix, you supply six parameters: the basic 5 (collection name, user name, user password, relation, view) along with the data itself. Each command has its own list of required and optional parameters, and they are shown in detail in Appendix A: Command Reference. The Quick Reference Card also provides a handy list of the commands and their parameters.

Note: the order of items in a parameter list is critical; you may not rearrange the order or the command will not work as expected.

Syntax alternatives

CallHelix supports three variations on the syntax of a command. This manual shows most commands using the syntax: CallHelix {parameter list} action command. A shorthand syntax is also available: use {parameter list} to command. Finally, for compatibility with CallHelix 1.0, it supports the old CallHelix {parameter list} selector ### syntax.

All three forms are functionally identical. Use the one you are most comfortable with.

Data format

Interaction with a Helix collection happens exclusively through views, in much the same way that you work with record data within Helix. In Helix, you open a collection, log in with a user name (supplying a password if needed), open a view, type data, and enter the record. When you add data via CallHelix, you must supply the same parameters. The Apple Event mechanism in Helix works through the collection/user/view model; your CallHelix commands need to know the name of the collection, the user (including the password), the view (including the relation in which the view appears), and the order of the fields on that view.

Data sent to Helix must be properly formatted for the view it is being sent to: the fields must be in the proper order (the view's tabbing order), the data must be the proper type (e.g., numbers in number fields), and the proper field delimiters must be used. The delimiter and error options specified in the view's Import and Export options dialog are used.

Building records with multiple fields

If a Helix record contains a single field of data the process is simple. But usually records are comprised of multiple fields. Typically the input data comes from discreet sources, perhaps from AppleScript dialog boxes, or by parsing a web page. Before you can send this data, you must **concatenate** your data into a single **string**. If this is not done correctly, Helix rejects the data just as if you had typed incompatible data on the view. In AppleScript the word tab is a predefined constant, equivalent to the tab character (ASCII Character 09). To send a record with multiple fields to Helix, you must create a single string of text consisting of multiple fields, with a tab inserted between each field. When you send that string to Helix, the tabs are treated the same as if you had imported the data into the view: they mark the point where one field ends and the next begins, exactly the same as if you were importing a text file or literally typing the data.

Concatenating data in AppleScript is done with the **&** operator. To format data that is

CallHelix works in much the same way that the Import and Export commands in Helix work.

By default, Helix uses the tab character as field delimiter and the return character as a record delimiter; if you change either of these settings on the corresponding view in Helix, you may have to modify your scripts to deal with the revised format the view uses for data exchange.

intended to be stored in consecutive fields on a view, concatenate the data as follows:

```
"sent" & tab & "from" & tab & "CallHelix" & tab & [etc...]
```

This puts the word “sent” in the first field, “from” in the second field, and so on.

Note: Unless you explicitly declare what type of data you want AppleScript to create, concatenated data may be concatenated into a **string** or a **list**. If you are unlucky, instead of the string ...

```
"sent→from→CallHelix→[etc...]"
```

... AppleScript converts the data to a list and you end up with ...

```
{"sent", tab, "from", tab, "CallHelix", tab, [etc...]}
```

... which causes an error when you send it to Helix.

To remove all doubt, **coerce** the data into string format by enclosing the whole string in parenthesis and adding the **as string** keyword to the end:

```
("sent" & tab & "from" & tab & "CallHelix" & tab & [etc...]) as string
```

Although CallHelix interprets field delimiter characters in this way, it does not treat record delimiters accordingly. If the view’s record delimiter is **return** and you send a return character in via CallHelix, the return character becomes part of the field it follows, exactly as if you were typing the record on a keyboard and typed a return (not Enter) character. This is consistent with Helix’s ability to store the record delimiter character within a field, but is an unexpected quirk in the way AppleEvent data is processed. Each record you intend to store in Helix must be sent as a distinct entity.

Handling replies from Helix

How Helix communicates with CallHelix: receiving data

Some commands – such as ones that retrieve records – return data to CallHelix. Data is returned in response to requests your script issues and it is your script’s responsibility to capture the data that is returned, storing that data in an AppleScript variable or otherwise acting on it. AppleScript has two commands that store data in a variable: **set** and **copy**.

Examples:

```
set myData to (CallHelix {"MyCollection", "MyUser", "MyPass", "MyRelation", "MyView"}) action retrieve records as string)
```

or

```
copy (CallHelix {"MyCollection", "MyUser", "MyPass", "MyRelation", "MyView"}) action retrieve records as string) to myData
```

Both of these lines give you the same result: the variable named `myData` containing the data Helix sends back, all of the records from the view ‘MyView’ in this example.

Why two forms? Good question. While there are subtle differences in the way AppleScript handles the two, a discussion of that is beyond the scope of this manual. For most purposes, either one is fine. Use whichever one you prefer.

Apple’s Script Editor (and most other script editors) have an **Event Log** window that can be used to observe the values that are returned as a script executes. Many statements generate log entries on their own, but by using the `log` command you can see the result of any statement. To see the result of the example above, you can use this line:

```
log myData
```

This displays the contents of `myData` in the Event Log window.

Handling errors

AppleScript’s try statement

All CallHelix commands return an error number and error message if the command is not successful. To handle errors, you enclose commands in a **try** block. A simple AppleScript **error handler** looks like this:

```
tell application "Helix Scripting"
  try
    CallHelix {"MyCollection", "MyUser", "MyPass", "MyRelation", "MyView"} action store one record
  on error errMsg number errNum
    display dialog errMsg & " (" & errNum & ")"
  end try
end tell
```

This script generates an error because `store one record` requires one more parameter: the data to store.

If you copy that short script to your script editor and run it right now, you will probably get

this error message: “An error occurred: Application isn’t running.” AppleScript is telling you that there is no Helix application running. A good thing about using a try statement to trap errors is that you can often figure out what the problem is simply by studying the error that is returned. Launch Helix RADE and run the script again. You should now get error number 20: “No collection opened with named Helix application.” In other words, Helix is launched, but no collection is open. To take it one step further, open a collection and run the script again. As long as it isn’t named “MyCollection” Helix returns the next logical error: number 30, the “Incorrect Collection Name” error. CallHelix returns errors in a logical manner, building up from the simple errors to the more complex.

Update Note: When Helix returns an error, it returns the error number as it should, but instead of an informative error message, it returns the text “An error of type *nn* has occurred” (*nn* being the error number) which is not at all informative. CallHelix 2 intercepts this and substitutes an informative message, so it is no longer necessary to use an error translation routine.

Error codes that originate in Helix are listed in Appendix B: Helix Apple Event Error Codes.

Handling errors gracefully

It is important to understand this about AppleScript: if you trap an error (using the try statement) the script jumps down to the `on error` section, executes the commands found in there, and then continues on with the script (unless an error is encountered within the `on error` code itself). If you do not **trap** an error, AppleScript stops execution of the current routine and sends information about the error back to the **caller**. If an error is not dealt with, it can leave **processes** (access channels) to Helix open. It is highly recommended – especially while you are learning to use CallHelix – that you check every call for errors. Learn to handle errors early on and your scripting experience will be much more pleasant. If you follow the little exercise above, you will notice that AppleScript displays a dialog informing you about the problem. The program doesn’t just quit: it **handles** the error gracefully. Writing good error handlers gives you the opportunity to recover and continue execution, or at the very least, allows you to clean up loose ends in your script.

Additional notes

View permissions defined in Helix are ignored when data is exchanged via Apple Events. You can retrieve data from views with **export** permission disabled, **store** data in views with **import** permission disabled, and delete records from views with **delete** permission disabled. This is considered a bug and is expected to change in a future version of Helix, so do not rely on this behavior.

Validations defined in Helix are respected when data is entered via Apple Events. If you try to **store** data in a view that does not meet the validation requirements for that relation, the data is rejected and Helix returns error 1000 (“An error was encountered trying to store data on an entry view”).

There is currently no provision for specifying queries in Helix via Apple Events. Any command that returns records can return every record the view can display, exactly the same as when “Export All” is used on that view from within Helix. For collections open with Helix RADE and Engine, the query that was last set by the user for the view is used. For collections open with Helix Server, the default query (the query that was set the last time the collection was opened in RADE or Engine) for the view is used.

Since Helix does not initiate Apple Events, there is no provision for maintaining referential integrity between Helix and CallHelix. In other words, the data received by CallHelix is cold as of the moment Helix sends the data.

Posts attached to a view’s **Data Entry** column happen when data is stored or deleted. Beginning with Helix 5.0, posts attached to a view’s **On Export** column are triggered when data is retrieved.

External access to a collection via CallHelix requires one client access point per open process. In Helix 4.x, a Helix Server with a 5 client license can not process Apple Events if there are 5 Helix Clients already visiting the collection. Helix RADE can have only one Apple event access active at any time. A 260 error (“Maximum number of users exceeded”) is returned if you exceed this limitation. In Helix 5.0–5.2, the maximum user check is disabled, so you do not see this error, regardless of how many processes are currently open. In Helix 5.3 and later, the maximum user check is reinstated with a limit that can be adjusted by the collection administrator.

Regardless of which version of Helix you are using, you should always remember to close processes when you are finished with them. Open processes draw CPU time away from other tasks, so leaving processes open can slow Helix down. Your error handling should take care to close processes that may have been taking place when an error occurs.

Helix and CallHelix version numbers

In CallHelix 2.0 and later, you can determine the exact version with CallHelix’s `CallHelixVersion`

command. See Appendix A: Command Reference for more information.

In Helix 5.2 and later, you can check the exact version of Helix with Helix's version command. See the Helix manual for more information.

Helix 5.3 added the *doMenu()* event, which when used in conjunction with CallHelix can be used to create very powerful AppleScript solutions.

Helix 6.0 added an OS X native version of Helix Server, enabling the OS X native components of CallHelix to work to their full potential.

Accessing multiple collections on one computer

Opening multiple collections on one computer requires you to run multiple copies of Helix. In this situation, the *target* parameter assures the correct copy of Helix is addressed. Since two collections with the same name can not be running on the same computer, sending commands to the wrong copy of Helix results in an "Incorrect Collection Name" error.

To keep CallHelix from attempting to send data to the wrong collection, use the *target* keyword followed by the name of the Helix application you want to communicate with:

```
CallHelix {"MyCollection", "MyUser", "MyPass", "MyRelation", "MyView"} action retrieve records as string target "Helix Engine"
```

This is useful when moving data from one collection to another on a single machine.

The target parameter can be used with all CallHelix commands, and data returned is identical to when it is not used.

Accessing collections on a remote computer

CallHelix can send commands to remote computers, opening up very powerful capabilities. For example, a single script can synchronize data in two collections running in different offices, or allow a web server to manipulate data in a Helix collection.

In prior versions of CallHelix, you accessed Helix applications running on a remote machine by talking to the Helix Scripting application running on that machine. CallHelix 2.1 improves on this, allowing you to bypass this intermediate step and access the Helix application directly. This results in faster response and removes the need for a secondary CallHelix license for use on the target machine.

See Appendix E: Targeting a Remote Computer for more information on targeting remote Helix applications.

About the examples

The following chapters provide explanation and examples demonstrating how to add, retrieve, and delete data in a Helix database. The code snippets work with the Helix collection "MyCollection" which is found in the "CallHelix Tutorial" folder of your Helix package. In the collection "MyCollection", there is a user named "MyUser" with a password "MyPass" and a relation called "MyRelation" containing view named "MyView".

To work with the examples, open "MyCollection" – it is a Helix version 5.0.2 collection, update it if necessary to work with your copy of Helix – and make sure it is in User Mode. Then launch Script Editor, open up a new script window, type (or copy/paste) the snippets into the editor window and run them.

For the sake of simplicity, AppleScript examples found in this section primarily use string values – variables are used only when required. Many simple scripts can be written without the need for variables, but if you want to tap into the real power of AppleScript, you must learn to work with variables. If you understand this already, it should be easy for you to follow the logic presented here. Sample code that makes use of more advanced AppleScript techniques is also included in the CallHelix package.

Chapter 5: Introduction to Complex Processes

The following sections build on the information presented in *Chapter 4: CallHelix Basics* on page 11. You should be comfortable with the concepts introduced there before proceeding.

Introduction

Moving beyond the simple access methods enables you to accomplish more, and to accomplish it more efficiently. The **complex processes** introduced here provide more power and flexibility, but they require more of you, the script writer.

About complex processes

Unlike the **simple processes**, which are self-contained and required just a single line of code, **complex processes** require that you set up and maintain a communication channel with Helix. This communication is called a **process** and is identified by its **ProcessID**. The flow of communication in a complex process is:

1. AppleScript requests a communication channel, informing Helix of the purpose of the access.
2. Helix opens a channel and returns a ProcessID the AppleScript can use to maintain the access.
3. AppleScript communicates through the assigned channel, performing tasks as needed.
4. Helix reports on the success of the tasks and returns data as requested.
5. AppleScript informs Helix that the channel can be closed.
6. Helix closes the channel.

It helps to think of this process as analogous to the steps you take when working directly in Helix: opening a view, working with the records, and closing the view when you are finished using it.

It is important to keep in mind that when you establish a process it is your responsibility to close the process when you are finished using it. Leaving processes open can have a detrimental effect on Helix's overall performance.

When to use complex processes

If you are adding multiple records to a database, it is more efficient to work via a complex process. Because a complex process opens and closes the communication channel as distinct steps, the actual process of adding multiple records is faster with a complex process, since it avoids the "opening and closing" of the view each time.

If you are retrieving a large amount of data from Helix, AppleScript can be overwhelmed by the amount of data if retrieved all at once via a simple retrieve process, since it must be stored in memory. In this case, working with a complex process is significantly more efficient. Accessing the field names on a view or the data in rectangles outside a repeat rectangle (a list) also requires the use of complex processes.

And since there is no simple process for deleting records, complex processes are *required* to delete data.

Chapter 6: Storing Records

This section builds on the information presented in *Chapter 4: CallHelix Basics* on page 11 & *Chapter 5: Introduction to Complex Processes* on page 16. You should be comfortable with the concepts introduced in those chapters before proceeding.

Introduction

There are two different types of store calls. The **store one record** process (See 'Adding records' in Chapter 4) is used for infrequent store requests and is an all-in-one call. The **store records** process (introduced here) requires that your script perform a series of steps, analogous to opening a view, adding records, and closing the view.

Store records should be used when adding multiple records, as it is significantly more efficient (and faster) than repeatedly calling **store one record**.

About complex storage

Complex storage requires your script to manage a communication channel with Helix. This process centers around a **ProcessID**, which Helix assigns when you initiate the complex storage request. Once this communication channel has been established, your script can send records to Helix. The conversation goes something like this:

AppleScript: I want to add records using the view "myView" in the relation "myRelation"

Helix: OK, here's the ProcessID to use.

AppleScript: Use that ProcessID to store this record ...

Helix: Done.

AppleScript: Now, use that ProcessID to store this record ...

Helix: Done

AppleScript: Close the process, I'm all done.

It is important to keep in mind that when you establish a process it is your responsibility to close the process when you are finished using it. Leaving processes open can have a detrimental effect on Helix's overall performance.

When to use complex storage

If you are adding multiple records to a database, it is more efficient to work via a complex process. Because a complex process opens and closes the communication channel as distinct steps, the actual process of adding multiple records is faster with a complex process, since it avoids the "opening and closing" of the view each time.

Complex storage: the process

Preparing to store: Create Process for Store

The first step is to open a communication channel (a **process**) and retrieving a **ProcessID** reply from Helix. This is done with the **create process for store** command. It is important to remember that **create process for store** does not store data – it opens a communication channel to a specific view in the collection and returns a **ProcessID** that you then use to send data to that view. To establish a process for storage, supply the **Basic 5** parameters:

```
set myProcessID to CallHelix {"MyCollection", "MyUser", "MyPass", "MyRelation", "MyView"} action create process for store
```

Once a ProcessID has been established, it is associated to the specified view. You can add data through that view at any time until the process is closed. (Just don't forget to close the process when you are finished using it.)

Create process for store only works with entry views. If you try to create a storage process with a list view, Helix returns error 200 (Not an entry view).

Sending data to Helix: Store Record

Store record works in conjunction with a ProcessID. Once you have acquired a ProcessID, all that is needed to add data is to supply CallHelix with the ProcessID and the data to store:

```
set theWholeRecord to ("Sent from" & tab & "CallHelix" & tab & "2.1")
set theResult to CallHelix {myProcessID, theWholeRecord} action store record
```

As with store one record, the data sent to Helix must be sent as a text string using the Helix view's field delimiter to break the data into distinct fields. Also, as with store one record, record delimiters are treated as a regular character (just as you can type a return character in a field on a view that uses the return character as its record delimiter):

You can send up to 10 records at a time to Helix, and the plural form can also be used:

```
set Record1 to ("Field 1" & tab & "Field 2") as string
set Record2 to ("Another record" & tab & "to add") as string
set theResult to CallHelix {myProcessID, Record1, Record2} action store records
```

Sending multiple records in a single command is very efficient, but the form shown above has a serious limitation: you must know exactly how many records are going to be stored. In cases where you do not know this, you can combine records as items in a list:

```
set Record1 to ("Field 1" & tab & "Field 2") as string
set Record2 to ("Another record" & tab & "to add") as string
set myRecordList to {Record1, Record2}
set theResult to CallHelix {myProcessID, myRecordList} action store records
```

It is perfectly legal to open a process and leave it open indefinitely, but in practice it is better to open a process, send your data, and close the process. You can have multiple processes active at the same time, but the number is limited by the version of Helix you are running and your Helix license.

Closing the process: Close Process

When you are finished sending data to a process, remember to close it:

```
CallHelix {myProcessID} action close process
```

It is important to keep in mind that when you establish a process it is your responsibility to close the process when you are finished using it. Leaving processes open can have a detrimental effect on Helix's overall performance.

Notes on storing records

Store records respects all field validations on the entry view. If data sent into Helix generates an error – whether a validation failure or a data type error – Helix returns error 1000 (“An error was encountered trying to store data on an entry view”) just as it would if you attempted to import the data from a text file. The **On Error** settings in the view’s import/export options dialog are also honored.

Store records can pass the view’s record delimiter through as regular data. Therefore, if the view’s record delimiter is the return character, sending ...

```
CallHelix {"MyCollection", "MyUser", "MyPass", "MyRelation", "MyView", "my" & return & "data"} action store one record
```

... results in a single record with the text "my...data" stored in the view’s first field, with the words ‘my’ and ‘data’ on two separate lines, just as if you typed the data on the view.

Store records triggers posts that are set in the view’s **On Entry** column.

The number of records Helix accepts in a single **store records** call is determined by the value stored in the Helix application’s HAEC resource. The default value is 10. See Appendix C: Helix HAEC Resources for more information on the HAEC resource.

Checking the success of the record storage

Store records returns a list of flags (true/false) that indicate whether the records were stored successfully or not. The items in the list correspond directly to the records sent to Helix. For example, if 5 records are successfully sent in a single call, the reply is:

```
{true, true, true, true, true}
```

This information can be used to confirm that the data was stored. In practice, we’ve never seen a false reply; Helix always returns an error code when a record can not be stored.

If any record is rejected an error (1000) is generated and no list is returned at all. Because of this, it is impossible to know how many of the records were stored successfully. It is therefore recommended that you send a single record with each command when it is critical to know that each record was stored.

Putting Store Records to work

All together, a simple **store records** section of code looks like this:

```
try -- open a view, add a single text field, and close a view.
  set myProcessID to 0 -- initialize the variable first
  set aRecord to ("Hello" & tab & "I'm in field 2!") -- concatenate data into a tab delimited string
  set myProcessID to (CallHelix {"MyCollection", "MyUser", "MyPass", "MyRelation", "MyView"} action create process for store) -- open process
  set theResult to CallHelix {myProcessID, aRecord} action store record -- send data
on error errorText number errorNumber -- trap an error, if one happens
  display dialog errorText -- display an alert dialog with the error message
end try
if (myProcessID > 0) then CallHelix {myProcessID} action close process -- Helix's ProcessIDs are always greater than 0
```

The first line of this code initializes the variable we use to store our ProcessID – it is always a good idea to initialize variables before using them. The second line creates a tab delimited string with two fields in it. The third line (create process for store) opens the communication channel and gets the ProcessID back from Helix and the fourth line (store record) sends the data to Helix. All of this is enclosed in a try block, and if we encounter an error the remaining lines in the try block are skipped, the code in the on error section is executed, and display dialog shows us what went wrong, and the script carries on from there.

The final step is to close the process with close process. If we didn't get a ProcessID from Helix, the value of myProcessID is still 0. Attempting to close a ProcessID that is not open generates another error, so we test myProcessID (using an if-then statement) and only call close process if we have a genuine ProcessID.

Again, remember the importance of closing every process you open.

Adding two records in two commands

The real power of store records is that we can store multiple records faster since the view is not opened and closed each time:

```
try -- illustrate sending two records via a single process
  set myProcessID to 0 -- initialize the variable first
  set myProcessID to (CallHelix {"MyCollection", "MyUser", "MyPass", "MyRelation", "MyView"} action create process for store)
  set aRecord to ("Hello" & tab & "Goodbye") as string -- concatenate data into a tab delimited string
  set theResult to CallHelix {myProcessID, aRecord} action store record -- store one record
  set aRecord to ("Another" & tab & "Record") as string -- we can (of course) reuse the variable.
  set theResult to CallHelix {myProcessID, aRecord} action store record
on error errorText number errorNumber
  display dialog errorText
end try
if (myProcessID > 0) then CallHelix {myProcessID} action close process -- Helix's ProcessIDs are always greater than 0
```

Adding two records in one command

We can get the same result with more efficient code by sending multiple records in a single command:

```
try -- illustrate sending two records via a single process
  set myProcessID to 0 -- initialize the variable first
  set myProcessID to (CallHelix {"MyCollection", "MyUser", "MyPass", "MyRelation", "MyView"} action create process for store)
  set Record1 to ("Hello" & tab & "Goodbye") as string -- concatenated record
  set Record2 to ("Another" & tab & "Record") as string -- another concatenated record
  set myRecordList to {Record1, Record2} -- combine two records into a list
  set theResult to CallHelix {myProcessID, myRecordList} action store records -- send a list of records to Helix
on error errorText number errorNumber
  display dialog errorText
end try
if (myProcessID > 0) then CallHelix {myProcessID} action close process -- Helix's ProcessIDs are always greater than 0
```

Adding *n* records, one by one

Another demonstration, this one creates 10 records, one at a time:

```
try -- add an AppleScript loop to make this add 10 records, two fields per record.
  set myProcessID to 0 -- initialize the variable first
  set myProcessID to (CallHelix {"MyCollection", "MyUser", "MyPass", "MyRelation", "MyView"} action create process for store)
  repeat with i from 1 to 10
    set theResult to CallHelix {myProcessID, "Record " & i & tab & (i as text)} action store record
    -- concatenate data & fields, coerce number i to text
  end repeat
on error errorText number errorNumber
  display dialog errorText
end try
if (myProcessID > 0) then CallHelix {myProcessID} action close process -- Helix's ProcessIDs are always greater than 0
```

Adding *n* records in one command

Because **store records** allows you to store multiple records in a single command. The previous script can be rewritten to take advantage of this, and make it more efficient:

```
try -- add an AppleScript loop to make this add 10 records, two fields per record.
  set myProcessID to 0 -- initialize the variable first
  set myProcessID to (CallHelix {"MyCollection", "MyUser", "MyPass", "MyRelation", "MyView"} action create process for store)
  set myList to {} -- initialize the list so we can work with it inside the loop
  repeat with i from 1 to 10
    copy ("Record " & i & tab & i) as string to the end of myList -- add this record to the end of the list named myList
  end repeat
  set theResult to CallHelix {myProcessID, myList} action store records -- send all 10 records at once
on error errorText number errorNumber
  display dialog errorText
end try
```

```
if (myProcessID > 0) then CallHelix {myProcessID} action close process -- Helix's ProcessIDs are always greater than 0
```

A single command can store any number of records, up to the limit set by Helix. The default limit is 10. See Appendix C: Helix HAEC Resources for more information on this limit.

Importing a text file into Helix

A common task in Helix is importing (loading) data from text files. Within Helix you use **Import** to do this, but CallHelix can replicate this function, giving you the power of AppleScript to pre-process the file before sending it to Helix. With AppleScript you can send the contents of a text file to more than one view, skip lines that contain information you do not want imported, and do many other things that are difficult or impossible in Helix.

Here is a simple AppleScript that imports data from a text file into a Helix view. Most of the error checking has been removed to keep it simple. An example with full error checking is included in the CallHelix package.

```
try -- one error handler to cover the main process
  set myProcessID to 0 -- initialize the variable
  set FilePath to (choose file with prompt "Pick a text file to import into Helix ..." of type {"TEXT"}) -- Calls the Mac OS file dialog
  set sourceFile to (open for access FilePath)
  set myProcessID to CallHelix {"MyCollection", "MyUser", "MyPass", "MyRelation", "myView"} action create process for store
  repeat
    try
      set thisLine to read sourceFile before return as text -- get a line, assumes CR for record delimiter
      set theResult to CallHelix {myProcessID, thisLine} action store record -- send this data to Helix
      if (theResult is false) then display dialog "A line was not stored for some reason."
    on error errTxt number errNum
      if (errNum ≠ -39) then -- don't bother showing the end-of-file error
        display dialog errTxt
      end if
      exit repeat -- get out of repeat, but keep going so we can close the process
    end try
  end repeat
on error errTxt number errNum
  display dialog errTxt
end try
close access (sourceFile) -- close the text file, we're done
if (myProcessID > 0) then CallHelix {myProcessID} action close process -- close the process, we're done
```

Chapter 7: Retrieving Records

This section builds on the information presented in *Chapter 4: CallHelix Basics* on page 11 & *Chapter 5: Introduction to Complex Processes* on page 16. You should be comfortable with the concepts introduced in those chapters before proceeding.

Introduction

Moving beyond the simple retrieval method described in Chapter 4 is essential if you are working with relations with large data sets. This chapter presents additional commands that enable you to take advantage of more powerful CallHelix retrieval capabilities.

There are many different ways to retrieve records. The easiest is the **retrieve records** process (described in Chapter 4) but it can only be used to return relatively small sets of records. **Complex retrieval** processes (described in this chapter) require you to manage the process in detail, but give great power to your scripts.

Between the simple **retrieve records** process and the **complex retrieval** processes described in this chapter is another option: **get partial view data**. This hybrid command provides some of the ease of retrieve records along with some of the power of complex retrieval. See Appendix A: Command Reference for information on this command.

About complex retrieval

Complex retrieval involves more steps than complex storage. As with complex storage, the process centers around a **ProcessID**, which Helix assigns when you send the complex retrieval request. Depending on the complexity of the query attached to the view, the number of records in the relation, and other factors gathering all of the required records can take time. Proper optimization techniques in Helix can speed this up just as it does in Helix. It is critical that your script wait until Helix has gathered the records before requesting them. (CallHelix 2 includes safeguards to make sure the view is ready before requesting records.) When Helix has signified that it is ready, your script can then ask for the records – in multiples of 10 or less. The conversation goes something like this:

```
AppleScript: I want to retrieve data from the view "myView" in the relation "myRelation"
Helix: OK, here's the ProcessID. I'll get to work on it.
AppleScript: Let me know when you are done.
Helix: Done. There are 600 records and the view uses Tab for the field delimiter and Return for the
record delimiter.
AppleScript: Give me 10 records, starting with the 1st one.
Helix: Here they are ...
AppleScript: Give me 10 more records, starting with the 11th one.
Helix: Here they are ...
AppleScript: Close the process, I'm done.
```

It is important to keep in mind that when you establish a process it is your responsibility to close the process when you are finished using it. Leaving processes open can have a detrimental effect on Helix's overall performance.

When to use complex processes

Although the main attraction of these commands is in their ability to provide fine control over the data returned, they are required when the number of records retrieved requires a significant amount of memory or retrieving the entire set of records is unnecessary.

Using complex retrieval processes also enables you to do many things the simple retrieval method can not do:

- Retrieve a subset of the view's records
- Retrieve the number of records on the view without retrieving the actual records
- Retrieve the delimiters used by the view
- Retrieve the field names on the view
- Retrieve header info (outer abaci) on list views
- Control the format of data returned from the views with subforms
- Access Helix's unique internal record identifiers

Complex retrieval: the process

Preparing to retrieve: Create Process for Retrieve

The first step is to open a communication channel (called a process) and retrieving a

ProcessID reply from Helix. This is done with the *create process for retrieve* command. To establish a process for retrieval, supply the **Basic 5** parameters:

```
set myProcessID to CallHelix {"MyCollection", "MyUser", "MyPass", "MyRelation", "MyView"} action create process for retrieve
```

It is important to remember that *create process for retrieve* does not return any data. It only opens a communication channel to a specific view in the collection and returns a **ProcessID** that you use to retrieve data from that view.

Once a process has been established, it remains linked to the specified view until the process is closed.

Waiting for the view: Get View Summary

Before you can get records from a view, you must wait until the view is ready to return data. All that is needed here is supply the ProcessID and request the summary info:

```
set myViewDetails to CallHelix {myProcessID} action get view summary
```

Failing to wait for a view to be ready can result in incorrect data being returned by Helix. You should always use *get view summary* to make certain the view is ready before attempting to get records. CallHelix includes built-in safeguards to avoid errors, but you should consider it your script's responsibility to test the view before requesting records from it.

When the view is fully prepared, *get view summary* returns a record of three items: record count, field delimiter, and record delimiter. You access these record items as you do any AppleScript record item:

```
set theRecordCount to (record count of myViewDetails) -- assigning a record item to a variable  
set {theFieldDelim, theRecDelim} to {field delimiter of myViewDetails, record delimiter of myViewDetails} -- capture two items at once
```

Start characters are not supported by Helix's Apple Event operations, regardless of the view's Import/Export options.

Getting data from Helix: Get View Data as ...

Get view data works in conjunction with an open process. Once your script has opened a process (and acquired a ProcessID), you can retrieve data from that view as long as the process remains open. To retrieve data, supply the ProcessID, and specify the type of data to get from the view, which records to retrieve, and which format you want them in:

```
set DataType to 2 -- 2 is the data type for the actual records.  
set StartRecord to 0 -- start with the 1st record (internally Helix uses 'zero based' counting).  
set NumberToGet to 10 -- you can retrieve up to 10 records at once from Helix.  
set myRecords to CallHelix {myProcessID, DataType, StartRecord, NumberToGet} action get view data as string
```

In this example, Helix returns first 10 records found on the specified view. There are many options when retrieving records, and the form of the data returned is shaped by those options. Those options are discussed in detail below.

Although it is legal to open a process and leave it open indefinitely, in practice it is better to open a process, retrieve your data, and close the process. When a process is opened, the set of records the view can return is 'frozen' (much like a cold form) so it is rather useless to leave the process open once you have retrieved the records – the data set for that particular process never changes.

Closing the process: Close Process

When you are finished sending data to a process, remember to close it.

```
CallHelix {myProcessID} action close process
```

It is important to keep in mind that when you establish a process it is your responsibility to close the process when you are finished using it. Leaving processes open can have a detrimental effect on Helix's overall performance.

Notes on retrieving records

Get View Data as ... commands work with both entry and list views.

Get View Data as ... commands trigger posts that are set in the view's **On Export** column.

Get View Data as ... commands respect record locking if posts are triggered. Unlike Helix's built-in exporting, which partially completes the process and then stops if a locked record is encountered, CallHelix retrieval checks every record *before* starting to retrieve data. If a locked record is found, Helix returns error 270 ("A record is write locked") and no data is retrieved.

Record retrieval options

Introduction

Helix's internal Apple Events support a limited subset of the options available through CallHelix 2. CallHelix 1.x simply wrapped a scripting language around those internal events, but CallHelix 2 extends those events, reducing the complexity of managing data when retrieving it from Helix. To keep the syntax of the various retrieve commands consistent (and to maintain compatibility with CallHelix 1.x) CallHelix uses a format that may not at first seem totally logical. Having a copy of the Quick Reference Card on hand can make writing scripts that retrieve data from Helix much easier.

The **Get View Data as ...** commands are the most complex commands available in CallHelix. The five required and two optional parameters give you a great deal of control over the format of data returned from Helix.

Get View Data as ... required parameter list

The first four required parameters are passed to CallHelix as a list, as in this example:

```
set myDataList to (CallHelix {ProcessID, DataType, StartRecord, NumberToGet} action get view data
```

It is critical that the list data be in the exact order specified. The meaning of each parameter, in order, is described next.

ProcessID

When a process is opened – by calling **create process for retrieve** – the ProcessID is returned. See **Preparing to Retrieve** above, for more detail.

DataType

From Helix's viewpoint, a view can contain three different types of data: **data rectangle names**, **view header data**, and **primary view data**. The data type parameter defines the type of data you want from the view. Helix expects this value to be an integer, so you must provide the number that corresponds with the data type you want:

- 0: Data rectangle names (actual field/abacus names)
This data type returns the data rectangle names, i.e. the same header data that is returned by an export file when the **Include Field Headers** option is checked. Data type 0 works with any view.
- 1: Data in the view header (data on a list view, outside the repeat rectangle)
This data type returns the same data that is returned in an export file when the **Include Outer Abaci** option is checked. Data type 1 only returns data from list views that have abaci placed outside the repeat rectangle. In all other views the reply is an empty list.
- 2: Data on a view (all other data requests)
This data type returns the primary data from the view, the same as is returned in an export file. Data type 2 works with any type of view. When in doubt, use this data type.

When retrieving data rectangle names (data type 0) or data in the view header (data type 1) the parameters for **start record** and **number to get** are irrelevant and should not be included. The correct syntax is:

```
CallHelix {myProcessID, myDataType} action get view data as string
```

When retrieving data rectangle names (data type 0) the outer abaci names are returned only if **Include Outer Abaci** is checked on the view in Helix.

StartRecord

The start record parameter tells Helix where in the found set of records to begin when returning data. Unlike working with Helix's regular interface, with CallHelix you are not limited to retrieving the first record and moving forward from there. CallHelix can return any record from any position within the data set at any time. This value must be an integer.

In Helix, records are counted beginning with 0, not 1. (This **zero based** counting method is standard in the world of C/C++ and Pascal programming and since Helix's Apple Events are written in C, CallHelix inherits this trait.) Be sure to keep in mind that if there are **n** records on a view, the last record is record **n-1**, not record **n**. Therefore, on a view with 5 records, record 0 is the 1st record, record 1 is the 2nd record, etc. through record 4, the 5th (and last) record.

Do not confuse this record numbering with Helix's internal record IDs. This count is merely the ordinal number of each record the view can return, whereas the internal record ID is the identifier Helix uses to maintain each record within the collection. The record IDs literally represent the order in which the records were added to the relation.

NumberToGet

The number to get parameter tells Helix how many records to return. Helix expects this

value to be an integer, and it must be less than or equal to the value stored in the Helix application's HAEC resource. See Appendix C: Helix HAEC Resources for more information on the HAEC resource.

Get View Data as ... required format option

The fifth (and final) required parameter for the *Get View Data as ...* commands is the formatting option. CallHelix 2 supports the following three formatting options:

as Helix data: Helix's native format

This formatting option returns the data in Helix's native reply format. Compared to the CallHelix 2 alternatives, it is awkward to work with. It is provided for backwards compatibility and we do not recommend using it in new scripts.

This format returns a list, the first item being the number of records returned, and the subsequent items alternating between the record ID and the record data (string delimited).

as string: each Helix record returned as a data delimited string

See 'Retrieve each record as a single entity (delimited data)' in Chapter 4 for a description of this format.

as list: each Helix record returned as a list of fields

See 'Retrieve each record as multiple entities (list data)' in Chapter 4 for a description of this format.

One of these formatting options must follow the *get view data as ...* command, as in:

```
CallHelix {ProcessID, DataType, StartRecord, NumberToGet} action get view data as string
```

Technically, the formatting option is part of the keyword. There is no actual *get view data* command. The literal CallHelix commands are *get view data as Helix data*, *get view data as string*, and *get view data as list*. A script will not compile if you write *get view data* and leave the formatting information off.

Get View Data as ... optional parameter list

The native Helix Apple Events also require a fifth item in the parameter list, but because its usefulness is limited only to views with subform data, CallHelix makes it optional.

Tabbing style

The tabbing style parameter controls how Helix identifies host and subform data. Helix expects this value to be either **true** or **false**. If omitted, CallHelix inserts the value **true**.

true: indent subform data with an extra leading field delimiter

This option causes records that are returned from subforms use a leading delimiter character to differentiate the subform records from the host view records. Nested subforms indent once for each layer of nesting.

false: do not indent subform data

This option causes all records to be returned with no extra leading delimiters, regardless of the subform level they are from.

Because Helix does not provide the name of the relation from which data is returned, there is no easy way to determine whether a particular record is from the host relation, or from a relation referenced by an embedded subform. The leading delimiters concept should be sufficient to identify each record's origin, but it is ultimately left to the script writer to determine which relation each returned record represents.

Earlier versions of Helix always uses a tab (ASCII 9) character for subform indenting. Beginning with Helix 5.3, Helix uses the field delimiter specified by the view.

An undocumented feature in Helix allows you to pass two tabbing style parameters, resulting in a double indentation of subform data. This feature is not guaranteed to work in future versions of Helix.

Get View Data as ... additional options

Each of the *get view data as ...* commands supports the following options ...

Internal record IDs

A unique feature of Apple Event access to Helix data is access to the internal record IDs used by Helix to track each record. CallHelix allows you to retrieve record IDs by adding the optional *with record ids* parameter. For example:

```
CallHelix {"MyCollection", "MyUser", "MyPass", "MyRelation", "MyView"} action retrieve records as string with record ids
```

CallHelix returns all of the data available through that view as a list of records, with record containing two items: helix record and record id. Using the sample collection supplied with CallHelix, this is the result:

```
{ {record id: 10, helix record:"Record 1→Field 2→1_"},
  {record id: 11, helix record:"Record 2→Field 2 in Record 2→2_"} }
```

In addition to the helix record item that is returned, each record also contains a record id item. Access the record id in each record by using a repeat loop:

```
repeat with each in the result
  get (record id of each)
end
```

The primary use of the record IDs is in deleting data. Using the record ID ensures that the record you intend to delete is actually deleted. They can also be used to determine the "historical time frame" of the data, since every record added to a relation is given a consecutive record ID. Beyond that, they are mostly a curiosity.

More about record IDs

When a relation is first created, the first record stored in it is assigned record ID 1. Each subsequent record is given the next higher record ID. The record IDs are never reused. Even if you delete every record in a relation, the next record added is assigned the next higher ID. Knowing this can enable you to work with the retrieved data in ways that are not directly possible in Helix.

You can therefore assume that if you have found, for example, the record with record ID 93, that the record assigned record ID 93 will always be that same record. If record ID 93 no longer exists, you know it has been deleted. Of course, there are many operations in Helix – including direct typing – that can replace data in a record. In some cases (e.g., when programmers use Option 4 posting) a record ID may be deleted, but the data in that record ID is moved to a new record, and it is therefore associated with a new record ID. The same is true if a relation's records are exported, deleted, and then re-imported; the data is identical, but the record IDs are completely different.

The important thing to remember is that if you intend to take advantage of access to the internal record IDs, you must be aware of the techniques the programmer uses in Helix to store and manipulate data in that relation. And be careful: you don't want to be working with the wrong records!

There is no published specification regarding internal record IDs. Their behavior could change in a future version of Helix.

Pre-Specified delimiters

Helix itself always returns data in delimited string format. When CallHelix returns records in list format, it first determines what the actual field and record delimiters are for a view. CallHelix then locates the delimiters in the Helix string and divides it into the distinct items. By default, CallHelix gets the delimiter data from Helix's view summary data. If you know in advance what the delimiters are, you can make the process slightly faster by pre-specifying the delimiters. The **delimiters** parameter must be followed by a list of exactly two items: the field delimiter and the record delimiter. Example:

```
CallHelix {"MyCollection", "MyUser", "MyPass", "MyRelation", "MyView"} action retrieve records delimiters {tab, return}
```

The data returned is the same whether you provide this information or not. The delimiters option simply tells CallHelix what the delimiters are so it can save time asking Helix what they are all over again.

Why bother? In a word: performance. If the **delimiters** parameter is not specified with a **get view data as ...** command, CallHelix internally performs another **get view summary** in order to make sure the view is ready and, in the case of **get view data as list**, to retrieve the record and field delimiters for the targeted view. Scripts that require maximum execution speed (CGIs, for example) should cache the delimiter results from **get view summary** and pass them back to CallHelix through the delimiters parameter.

It is also possible to 'trick' CallHelix into dividing up the Helix data into a list based on some other delimiter than the one that is set on the view. For example, if you tell CallHelix that the field delimiter is a space character, and CallHelix returns each word in the record as a separate list item.

Notes on retrieving records

Beginning with Helix 5.0, **retrieving records** triggers posts that are set in the view's **On Export** column.

Records can be retrieved from both entry and list views.

View permissions set in Helix for the chosen user do not prevent Apple event based

deletions. A user with export permission removed on a view can still retrieve records through CallHelix. This is a bug in Helix that should be addressed in a future version.

The number of records Helix can accept per call is controlled by the value stored in the Helix application's HAEC resource. See Appendix C: Helix HAEC Resources for information on the HAEC resource.

Putting Get View Data As ... to work

Here is a basic AppleScript that illustrates the entire complex retrieve process start to finish, writing the data (including field headers) to a text file, just like a Helix export does. Detailed error checking is omitted to keep the example clear.

```
try -- one error handler to rule them all
  set myProcessID to CallHelix {"MyCollection", "MyUser", "MyPass", "MyRelation", "MyView"} action create process for retrieve
  set myViewDetails to CallHelix {myProcessID} action get view summary
  set theRecordCount to (record count of myViewDetails)
  display dialog "There are " & theRecordCount & " records to retrieve." -- just to see what we got

  tell application "Finder"
    activate
    set OutputFilePath to (choose file name with prompt "Save Helix Data ..." default name "CallHelix Output.text")
    set theFile to (open for access (OutputFilePath as string) with write permission)
    set eof of theFile to 0 -- empty out the old file, if we are replacing one
  end tell

  set HeaderType to 0 -- Get header data (field names)
  set theData to (CallHelix {myProcessID, HeaderType} action get view data as string)
  set theHeader to helix record of item 1 of theData -- getting header, there is only one item
  tell application "Finder" to write theHeader to theFile

  tell me to activate
  set DataType to 2 -- Get inner repeat data starting with record '0', 10 at a time
  set StartRecord to 0
  set NumberToGet to 10
  repeat until (StartRecord > theRecordCount) -- keep repeating until we've dealt with every record.
    set thisData to (CallHelix {myProcessID, DataType, StartRecord, NumberToGet} action get view data as string) -- get 10 records
    repeat with i from 1 to (count items in thisData) -- step through records. Be sure to count items, we don't know there are 10
      set thisRecord to (helix record of item i of thisData) -- get record out of each list item
      tell application "Finder" to write thisRecord to theFile -- write it out
    end repeat
    set StartRecord to StartRecord + 10 -- increment the counter or we get the same records over and over
  end repeat
on error errTxt number errNum
  display dialog errTxt
end try

try
  tell application "Finder" to close access theFile -- close the file we were writing to, or the Finder will be unhappy
on error errTxt number errNum
  display dialog errTxt
end try

try
  CallHelix ProcessID action close process-- put this in a separate try in case something happens (like the user cancels)
on error errTxt number errNum
  display dialog errTxt
end try
```

Chapter 8: Deleting Records

This section builds on the information presented in Chapter 4: CallHelix Basics & Chapter 5: Introduction to Complex Processes. You should be comfortable with the concepts introduced in those chapters before proceeding.

Introduction

There is no simple process for deleting records. The only way to delete records directly is via a *complex delete process*.

About deleting records

Deleting records in Helix requires that you work directly with Helix's internal RecordIDs. A RecordID is the internal number that Helix assigns to each record as it is created. RecordIDs are never reused and are not subject to arbitrary change. However since the internal RecordIDs are not normally available to the user, you should not rely on them to remain constant unless you are fully aware of the design methods used in the collection. Some Helix databases are constructed in a way that modifying records is done by a delete/create process (typically via Option 4 posting) and what may appear (by looking at the external data) to be the same record is actually a different one internally. The same is true if a relation's records are exported, deleted, and then re-imported—the data is identical, but the record IDs are completely different. Since RecordIDs are not reused, accidentally deleting the wrong data is unlikely to happen. Nonetheless, you must assume full responsibility for ensuring that you are deleting the correct records when working with the internal RecordIDs.

Make sure you are comfortable with the process required to retrieve records before attempting to delete records. Also make sure that you understand how to extract the record id item from the data that is returned in response to the *with record ids* option.

When deleting records, the conversation goes something like this:

AppleScript: Get the records from the view "myView" in the relation "myRelation" and include with the record IDs. (Using the simple retrieve process.)

Helix: OK, here are the records.

AppleScript: Now, get a ProcessID for the view "myView" in the relation "myRelation"

Helix: OK, here it is.

AppleScript: I want to delete the 3rd record in the list I got. Here is the ProcessID to use and the RecordID for the record to delete.

Helix: Deleted.

AppleScript: Close the process, I'm done.

It is not necessary to first retrieve records, as this example does, before deleting them. However, it is strongly recommended, as there is no other foolproof way to know the record IDs of the records you intend to delete.

Unlike the retrieval process, the view you use does not control the availability of specific records for deletion. (I.e. whether or not a record meets the query criteria for that view is not taken into consideration.) All records in a relation are able to be deleted through any view, regardless of whether they can be retrieved through that view.

Post operations attached to the view's "On Enter" column are triggered when a record is deleted, just as they are when the record is deleted from Helix itself.

Delete records: the process

Preparing to delete: Create Process for Retrieve

Because you typically need to retrieve the record IDs before you can delete them, the command to open a communication channel is the one used to create a retrieval process. (Review Chapter 7: Retrieving Records for information on creating retrieval processes.)

```
set myProcessID to CallHelix {"MyCollection", "MyUser", "MyPass", "MyRelation", "MyView"} action create process for retrieve
```

Once a ProcessID has been established, it is associated to the specified view. You can send data to that view as long as the process remains open.

Deleting: Delete Records

Delete records works in conjunction with a ProcessID and recordIDs. When retrieving records, use the *with record ids* option to get the record IDs for the records you intend to

delete. You can use any of the *get view data as ...* commands to get records and record IDs from the view.

-- Note: this sample uses literal values for retrieval parameter list. Typically you use variables.
set first10Records to (CallHelix {myProcessID, 2, 0, 10} action get view data as string with record ids) -- just do 10, because it's just an example!

Once you have a recordID, you delete its record by sending the recordID to CallHelix along with the ProcessID:

```
set theTargetRecord to (item 3 of the first10Records)
```

-- 'item 3' is just a randomly selected record, in a real life scenario, you want to apply more intelligence.

```
set thisRecordID to (record id of theTargetRecord)
```

```
CallHelix {myProcessID, thisRecordID} action delete record
```

Helix replies with an integer indicating the number of records successfully deleted. Check the result to confirm that it worked:

```
if (theResult is not equal to 1) then display dialog "The record was not deleted!" -- a failure of some sort occurred.
```

Alternatively, you can delete up to 10 records at a time. (The plural form *delete records* can also be used.) To delete more than one record at a time, you pass the ProcessID and the list of recordIDs to Helix:

```
set theDeletingList to {}
```

```
repeat with thisRecord in the first10Records
```

```
    copy (record id of thisRecord) to the end of theDeletingList
```

```
end repeat
```

```
CallHelix {myProcessID, theDeletingList} action delete records
```

Helix replies with an integer indicating the number of records successfully deleted. If every record was not deleted, the result is the number of records that were deleted. For example, if you sent 10 recordIDs for deletion and the result is 7, you know that the 8th (and all subsequent) record in your list was not deleted. Because of the extra work required to trap this type of error, it is recommended that unless you are deleting hundreds of records at once, you should send just one recordID at a time to Helix.

Closing the process: Close Process

When you are finished deleting data, remember to close the process.

```
CallHelix {myProcessID} action close process
```

It is important to keep in mind that when you establish a process it is your responsibility to close the process when you are finished using it. Leaving processes open can have a detrimental effect on Helix's overall performance.

Notes on deleting records

The singular (*delete record*) and plural (*delete records*) forms can be used interchangeably.

Delete records triggers posts that are set in the view's **On Entry** column.

Delete records can delete records from both entry and list views.

View permissions set in Helix for the chosen user do not prevent Apple event based deletions. A user with delete permission removed on a view is still able to delete records through CallHelix. This is a bug in Helix that should be addressed in a future version.

The number of records Helix accepts in a single **delete records** call is determined by the value stored in the Helix application's HAEC resource. See Appendix C: Helix HAEC Resources for more information on the HAEC resource.

In Helix versions prior to 5.0, the record locking that prevents a Helix view from deleting a record that another view is editing (i.e. is write locked) does not prevent an Apple Event delete. Through Apple Events you can delete a record while a Helix user is in the process of modifying it. This causes an index error that Helix Utility can detect and correct. This is fixed in Helix versions 5.0 and later.

Putting Delete Records to work

Here is a simple routine that deletes every record in a relation:

```
try -- one error handler to cover everything
```

```
    set myProcessID to CallHelix {"MyCollection", "MyUser", "MyPass", "MyRelation", "MyView"} action create process for retrieve
```

```
    set myViewDetails to CallHelix {myProcessID} action get view summary
```

```
    set theRecordCount to (record count of myViewDetails)
```

```
    set theDeletedRecordCount to 0 -- keep track of deletions
```

```
    set DataType to 2
```

```
    set StartRecord to 0
```

```
    set NumberToGet to 10
```

```

repeat until (StartRecord > theRecordCount) -- keep repeating until we've dealt with every record.
  set thisData to (CallHelix {myProcessID, DataType, StartRecord, NumberToGet} action get view data as string with record ids)
  repeat with each in thisData -- step through records. Be sure to count items, we don't know there are 10
    set thisRecordID to (record id of each) -- get record ID of each item
    set theResult to (CallHelix {myProcessID, record id of each} action delete records)
    set theDeletedRecordCount to (theDeletedRecordCount + theResult) -- keep track of successes
  end repeat
  set StartRecord to StartRecord + 10 -- increment the counter or we get the same records over and over
end repeat
display dialog (theDeletedRecordCount as string) & " of " & (theRecordCount as string) & " records on the view were deleted."
on error errTxt number errNum
  display dialog errTxt
end try
try -- close process needs to be separate in case something happens
  CallHelix myProcessID action close process
on error errTxt number errNum
  display dialog errTxt
end try

```

Chapter 9: Check View State

Introduction

Unlike the other CallHelix commands, **check view state** is not used to manipulate data in Helix. Its purpose is to test the validity of a setup prior to using it. For the most part it is not needed since you can learn the same thing by calling **create process for store** and trapping the error that occurs if the view is a list. (Error 200: Can not store data on a list view.)

When to use Check View State

You use check view state to find out if a view is an entry view or a list view.

To use **check view state** you supply the "basic 5" parameters (see Chapter 4), plus an additional error checking parameter (**true** or **false**) to indicate whether you want to know if the view is a list or not. With the parameter set to true, Helix returns error 200 (Can not store data on a list view) if the view is a list.

```
CallHelix {"MyCollection", "MyUser", "MyPass", "MyRelation", "myView", true} action check view state -- "true" turns on type checking
```

If one of the other parameters is not valid (e.g., there is no user that matches the supplied user name) Helix returns the standard error for that problem (No such user, Invalid password, etc.). It is your responsibility to trap these errors and handle them appropriately.

Putting Check View State to work

The following code fragments illustrate the four possible combinations of view type and check view's return error parameter.

```
try
  CallHelix{"MyCollection", "MyUser", "MyPass", "MyRelation", "myView", false} action check view state -- "false" returns no error for entry views
  display dialog "Entry view & false = no error."
on error errTxt number errNum
  display dialog errTxt
end try

try
  CallHelix{"MyCollection", "MyUser", "MyPass", "MyRelation", "ListView", false} action check view state -- "false" returns no error for list views
  display dialog "List view & false = no error."
on error errTxt number errNum
  display dialog errTxt
end try

try
  CallHelix{"MyCollection", "MyUser", "MyPass", "MyRelation", "myView", true} action check view state -- "true" returns no error for entry views
  display dialog "Entry view & true = no error."
on error errTxt number errNum
  display dialog errTxt
end try

try
  CallHelix{"MyCollection", "MyUser", "MyPass", "MyRelation", "ListView", true} action check view state -- "true" returns error 200 for list views
  display dialog "List view & true = error 200." -- you won't see this because of the error!
on error errTxt number errNum
  display dialog errTxt
end try
```

Appendix A: Command Reference

This section details the parameters and return values of each command.

How to read this command reference

Each entry starts with the command (and its native selector number) and a short description of its function. Following that is the **Required Parameter List** and the **Optional Parameter List**. Some items in these lists start with a number, which indicates the position this parameter must be in *inside the parameter list between the **CallHelix** and **action** keywords*. Parameters that do not start with a number go *after the **action** keyword*.

Reply describes the data that Helix returns to your script, assuming no error occurs.

Syntax Examples are simple statements showing what typical script command lines look like.

Reply Examples show what typical replies from Helix look like.

Notes covers typical issues and ‘gotchas’ that may be encountered.

Storing Data

store one record

Selector 110

Store a single record in database

Required Parameter List:

1. Collection Name
2. User Name
3. Password
4. Relation Name
5. View Name
6. Data to be stored (a record)

Optional Parameter List:

target

Reply:

{success/failure flags} (a list of one item)

Syntax Example:

CallHelix {CollectionName, UserName, UserPassword, RelationName, ViewName, theData} action store one record

Reply Example:

{true} (a list of 1 boolean)

Notes:

Data must be formatted as a field delimited text string, with no record delimiter. Start characters are not supported by Helix’s Apple Event operations, regardless of the view’s Import/Export options, and should not be included in the text string.

Attempting to pass a record delimiter results in the delimiter being stored in the record.

If a validation error prevents the record from being stored, error 1000 (data could not be entered) is returned. There is currently no way to retrieve the validation message from Helix.

create process for store

Selector 111

Establish a ProcessID for the purpose of storing records

Required Parameter List:

1. Collection Name
2. User Name
3. Password
4. Relation Name
5. View Name

Optional Parameter List:

target

Reply:

ProcessID

Syntax Example:

CallHelix {CollectionName, UserName, UserPassword, RelationName, ViewName} action create process for store

Reply Example:

57 (an integer)

Notes:

Be sure to call **close process** when finished with the ProcessID.

store record(s)

Selector 112

Store one or more records in the database via a ProcessID

Required Parameter List:

1. ProcessID (acquired via **create process for store**)
2. Data to be stored (1 to 10 records)

Optional Parameter List:

{record1, record2, record3, ...} up to 10 records total, enumerated individually, or combined as a list
target

Reply:

{success/failure flags} (a list of one or more items)

Syntax Examples:

CallHelix {ProcessID, Record} action store record -- a single record

CallHelix {ProcessID, Record1, Record2, ...} action store records -- enumerated items

CallHelix {ProcessID, {Record1, Record2, ...}} action store records -- combined as a list

Reply Example:

{true, true, ...} (a list of booleans, one per record submitted)

Notes:

Be sure to call **close process** when finished with the ProcessID.

The singular (store record) and plural (store records) forms can be used interchangeably.

Attempting to delineate records by concatenating data (with record delimiters between the records) results in the delimiter being stored in the last field and the subsequent data 'wrapping around' to the top field.

If a validation error prevents a record from being stored, error 1000 (An error was encountered trying to store data on an entry view set to stop on errors) is returned. There is currently no way to retrieve the validation message from Helix.

When an error (such as a validation error) occurs, the standard success/failure flags are not returned. (The error code is returned instead.) Consequently, if an error occurs while storing multiple records in a single call, there is no way to know how many of the records were actually stored. You may want to avoid sending multiple records in a single call, using a loop to send each record individually instead.

Retrieving Data

retrieve records as string

Selector 104

Get every record the view can display, as delimited data

Required Parameter List:

1. Collection Name
2. User Name
3. Password
4. Relation Name
5. View Name

Optional Parameter List:

with record ids
target

Reply: List of record, each containing one or two fields.

helix record (always)
record id (if 'with record ids' parameter is used)

Syntax Example:

CallHelix {CollectionName, UserName, UserPassword, RelationName, ViewName} action retrieve records as string

CallHelix {CollectionName, UserName, UserPassword, RelationName, ViewName} action retrieve records as string target "Helix RADE"

CallHelix {CollectionName, UserName, UserPassword, RelationName, ViewName} action retrieve records as string with record ids

Reply Example (using with record ids parameter):

{helix record: "DataRectangle1→DataRectangle2....", record id: 57}, {helix record: "DataRectangle1→DataRectangle2....", record id: 3}, ... }

Notes:

Each Helix record ends with the view's selected record delimiter. You can suppress the record delimiter by deselecting it on the view.

retrieve records as list

Selector 105

Get every record the view can display, as list data

Required Parameter List:

1. Collection Name
2. User Name
3. Password
4. Relation Name
5. View Name

Optional Parameter List:

with record ids
target

Reply: List of record, each containing one or two fields:

helix record (always)
record id (if 'with record ids' parameter is used)

Syntax Examples:

CallHelix {CollectionName, UserName, UserPassword, RelationName, ViewName} action retrieve records as string

CallHelix {CollectionName, UserName, UserPassword, RelationName, ViewName} action retrieve records as string with record ids

Reply Example (using with record ids parameter):

```
{{helixrecord: {DataRectangle1, DataRectangle2, ...} ", recordid: 57}, {helixrecord: {DataRectangle1, DataRectangle2, ...} ", recordid: 3}, ...}
```

Notes:

The field and record delimiters are automatically stripped from the reply.

create process for retrieve

Selector 100

Establish a ProcessID for the purpose of retrieving records

Required Parameter List:

1. Collection Name
2. User Name
3. Password
4. Relation Name
5. View Name

Optional Parameter List:

target

Reply:

ProcessID

Syntax Examples:

CallHelix {CollectionName, UserName, UserPassword, RelationName, ViewName} action create process for retrieve

CallHelix {CollectionName, UserName, UserPassword, RelationName, ViewName} action create process for retrieve target "Helix Engine"

Reply Example:

57 (an integer)

Notes:

Be sure to call **close process** when finished with the ProcessID.

When a view is opened in Helix, there is a pause between opening the view and when the list appears. That gap (however slight) is the time required for Helix to gather the appropriate records and sort them according to the chosen sort order. Never attempt to retrieve records from a view until it is ready. The **get view summary** command waits until the view is fully prepared, and should be called between creating a retrieval process and retrieving records. Query optimization minimizes this wait, just as it does in Helix itself.

get view data as Helix data

Selector 140

Retrieve a group of records in native format

Required Parameter List:

1. ProcessID (acquired via **create process for retrieve**)
2. Data Type (0-2)
3. Starting (ordinal) record to get (for data type 2 requests only)
4. Number of records (up to 10) to get (for data type 2 requests only)

Optional Parameter List:

5. Subform level tabbing (for data type 2 requests only)
- target

Reply:

The reply contains a list of data, ordered as follows: the first list element is the number of records in this

specific list. The remaining elements alternate between the recordID and the requested type's data in string format. Each data type returns different data from the view:

Data Type 0: Data Rectangle Names

- Item 1 contains the number 1 (the number of items in this list).
- Item 2 contains the names of data rectangles, in delimited string format.

Data Type 1: Data in the View Header

- Item 1 contains the number 1 (the number of items in this list).
- Item 2 contains data outside a list view's repeat rectangle, in delimited string format.

Data Type 2: Data on the View

- Item 1 contains the number of items in this list.
- Item 2-n contain the record ids (as integer) and record data (as text) in alternating items.

Syntax Examples:

```
CallHelix {ProcessID, DataType} action get view data as Helix data -- DataType = 0
CallHelix {ProcessID, DataType} action get view data as Helix data -- Data Type = 1
CallHelix {ProcessID, DataType, StartRecord, NumberToGet} action get view data as Helix data -- Data Type = 2
CallHelix {ProcessID, DataType, StartRecord, NumberToGet, false} action get view data as Helix data -- Data Type = 2, subform tabbing off
```

Reply Examples:

```
DataType 0: {{1, "Field1Name→Field2Name...↵"}}
DataType 1: {{1, "OuterAbacus1→OuterAbacus2...↵"}}
DataType 2: {7, 57, "DataRectangle1→DataRectangle2...↵", 3, "DataRectangle1→DataRectangle2...↵", ... }
```

Notes:

This is the original CallHelix command, and is maintained solely for compatibility with scripts written with CallHelix 1.x. It is not recommended for new scripts.

Be sure to call **close process** when finished with the ProcessID.

Data type 1 only returns data if the view being called is a list view and there are data rectangles outside the repeat rectangle on the view.

Data type 2 returns data for any record that can be seen on an entry view and for any record within the repeat rectangle on a list view.

The data returned by Helix includes the record delimiter.

If a target record is deleted (by another process or user) before it is retrieved, the record data consists of a single byte (ASCII Number 3).

get view data as string

Selector 141

Retrieve a group of records as Helix delimited string data

Required Parameter List:

1. ProcessID (acquired via **create process for retrieve**)
2. Data Type (0-2)
3. Starting (ordinal) record to get (for data type 2 requests only)
4. Number of records (up to 10) to get (for data type 2 requests only)

Optional Parameter List:

5. Subform level tabbing (for data type 2 requests only) with record ids (for data type 2 requests only) delimiters (acquired via **get view summary** – for data type 1 & 2 requests only) target

Reply: List of records, each record as follows:

Data Type 0: Data Rectangle Names

helix record: Names of data rectangles, as delimited text.

Data Type 1: Data in the View Header

helix record: Data outside a list view's repeat rectangle, as delimited text.

Data Type 2: Data on the View

helix record: Record data, as delimited text.
record id: Helix internal recordID (if **with record ids** is specified)

Syntax Examples:

```
CallHelix {ProcessID, DataType} action get view data as string-- DataType = 0
CallHelix {ProcessID, DataType} action get view data as string -- Data Type = 1
CallHelix {ProcessID, DataType, StartRecord, NumberToGet} action get view data as string -- Data Type = 2
CallHelix {ProcessID, DataType, StartRecord, NumberToGet} action get view data as string with record ids -- Data Type = 2
```

Reply Examples:

```
DataType 0: {{helix record: "Field1Name→Field2Name...↵"}}
DataType 1: {{helix record: "OuterAbacus1→OuterAbacus2...↵"}}
DataType 2: {{helix record: "DataRectangle1→DataRectangle2...↵"
, record id: 57}, {helix record: "DataRectangle1...↵", record id: 3}, ... }
```

Notes:

Be sure to call **close process** when finished with the ProcessID.

Data type 1 only returns data if the view being called is a list view and there are data rectangles outside the view's repeat rectangle.

Data type 2 returns data for any record that can be seen on an entry view and for any record within the repeat rectangle on a list view.

The record data returned by Helix includes the record delimiter.

If a target record is deleted (by another process or user) before it is retrieved, the record data consists of a single byte (ASCII Number 3).

get view data as list

Selector 142

Retrieve a group of records as list data

Required Parameter List:

1. ProcessID (acquired via **create process for retrieve**)
2. Data Type (0-2)
3. Starting (ordinal) record to get (for data type 2 requests only)
4. Number of records (up to 10) to get (for data type 2 requests only)

Optional Parameter List:

5. Subform level tabbing (for data type 2 requests only) with record ids (for data type 2 requests only) delimiters (acquired via **get view summary**) target

Reply: List of records, each record as follows:

Data Type 0: Data Rectangle Names

helix record: Names of data rectangles, each rectangle as its own list item.

Data Type 1: Data in the View Header

helix record: Data outside a list view's repeat rectangle, each rectangle as its own list item.

Data Type 2: Data on the View

helix record: Record data, each rectangle as its own list item.

record id: Helix internal recordID (if **with record ids** is specified)

Syntax Examples:

```
CallHelix {ProcessID, DataType} action get view data as list -- DataType = 0
CallHelix {ProcessID, DataType} action get view data as list -- Data Type = 1
CallHelix {ProcessID, DataType, StartRecord, NumberToGet} action get view data as list -- Data Type = 2
CallHelix {myProcessID, DataType, StartRecord, NumberToGet} action get view data as list delimiters {tab, return} -- delimiters specified
```

Reply Examples:

```
DataType 0: {{helix record: {Field1Name, Field2Name, ...}}}
DataType 1: {{helix record: {OuterAbacus1, OuterAbacus2, ...}}}
DataType 2: {{helix record: {DataRectangle1, DataRectangle2, ...}, record id: 57}, {helix record: {DataRect1, DataRect2, ...}, record id: 3}, ...}
```

Notes:

Be sure to call **close process** when finished with the ProcessID.

Data type 1 only returns data if the view being called is a list view and there are data rectangles outside the view's repeat rectangle.

Data type 2 returns data for any record that can be seen on an entry view and for any record within the repeat rectangle on a list view.

If a target record is deleted (by another process or user) before it is retrieved, the record data consists of a single byte (ASCII Number 3).

get partial view data (as list)

Selector 143

Retrieve a group of records as list data, assuming data type 2

Required Parameter List:

1. ProcessID (acquired via **create process for retrieve**)

Optional Parameter List:

2. Starting (ordinal) record to get
3. Number of records to get (any number, up to the number of records available to the view)
4. Subform level tabbing with record ids delimiters (acquired via **get view summary**) target

Reply:

helix record: Record data, each rectangle as its own list item.
record id: Helix internal recordID (if **with record ids** is specified)

Syntax Examples:

CallHelix {ProcessID, StartRecord, NumberToGet} action get partial view data
CallHelix {ProcessID, StartRecord, NumberToGet} action get partial view data with record ids

Reply Example:

{{helix record: {DataRectangle1, DataRectangle2, ...}, record id: 57}, {helix record: {DataRect1, DataRect2, ...}, record id: 3}, ...}

Notes:

Be sure to call **close process** when finished with the ProcessID.

If the second and third parameters are not specified the complete data set is returned.

If the third parameter is not specified, all records from the ordinal position specified in the second parameter through the end of the data set are returned.

If the second parameter is greater than the total number of records on the view, error 180 (End of List Encountered) is returned.

If a target record is deleted (by another process or user) before it is retrieved, the record data consists of a single byte (ASCII Number 3).

Deleting Data

delete record(s)

Selector 150

Delete records from the database

Required Parameter List:

1. ProcessID
2. Record ID of records to be deleted (1 to 10 records)

Optional Parameter List:

{record1, record2, record3, ...} up to 10 records total, enumerated individually, or combined as a list
target

Reply:

number of records deleted

Syntax Examples:

CallHelix {ProcessID, RecordID} action delete record -- a single record
CallHelix {ProcessID, RecordID1, RecordID2, ...} action delete records -- enumerated items
CallHelix {myProcessID, {RecordID1, RecordID2, ...}} action delete records -- combined as a list

Reply Example:

7 (an integer, representing the number of records deleted)

Notes:

The singular (delete record) and plural (delete records) forms can be used interchangeably.

Be sure to call **close process** when finished with the ProcessID.

According to the Helix specification, you should be able to check the reply against the number of RecordIDs sent to determine if all records were deleted, and if they are not equal, a problem was encountered. In practice **delete records** always returns an error when a problem is encountered.

Complex Process Support (Commands with a ProcessID)

close process

Selector 130

Release a ProcessID when finished using it.

Required Parameter List:

1. ProcessID

Optional Parameter List:

target

Reply:

none

Syntax Example:

CallHelix {ProcessID} action close process

Notes:

It is essential to close every process you open. Failure to do so can result in performance degradation in the Helix database.

get view summary

Selector 121

Wait until a view is ready and return the view's record count and delimiters.

Required Parameter List:

ProcessID

Optional Parameter List:

target

Reply: a record with three items:

record count: total number of records this view can access

field delimiter

record delimiter

Syntax Example:

```
CallHelix {ProcessID} action get view summary
```

Reply Example:

```
{record count: 343, field delimiter: tab, record delimiter: return}
```

Notes:

An unoptimized view can take a long time to return view summary data. You may need to use AppleScript's with timeout of n seconds to avoid script errors.

Early versions of Helix return the delimiters that were last selected in the view's import/export options dialog, even if they are currently deselected. This is fixed in Helix 5.2.

check view state

Selector 190

Test a view to find out if it is an entry or list view.

Required Parameter List:

1. Collection Name

2. User Name

3. Password

4. Relation Name

5. View Name

6. Check View Type

Optional Parameter List:

target

Reply:

none (or error 200)

Syntax Example:

```
CallHelix {CollectionName, UserName, UserPassword, RelationName, ViewName, true} action check view state -- "true" turns on type checking
```

Reply Example:

```
n/a
```

Notes:

When the **check view type** parameter is set to true, the command tests the view to determine if it is a list or not. (Passing false in this parameter is virtually pointless.) If the view is a list, error 200 (Not an entry view) is generated, otherwise no reply is given. Therefore, **check view type** must be enclosed in a try statement to be effective.

If one of the other parameters is not valid (e.g., there is no user that matches the supplied user name) Helix returns the standard error (No such user, Invalid password, etc.) for that problem.

test view for readiness

Selector 120

Test a view to find out if it is ready (legacy call)

Required Parameter List:

1. ProcessID

Optional Parameter List:

none

Reply:

If the view is not ready: {false}

If the view is ready: {true, number of records, field delimiter, record delimiter}

Syntax Example:

```
CallHelix {ProcessID} action test view for readiness
```

Reply Examples:

```
not ready: {false}
```

ready: {true, 583, tab, return}

Notes:

This is the original Helix call, and is maintained solely for compatibility with scripts written with CallHelix 1.x. It is not recommended for new scripts. New scripts should use ***get view summary*** instead.

Since you can not proceed until Helix is ready, you must repeatedly check the process until the first item in the list is true. An AppleScript repeat loop set to repeat until item 1 of the result is true should be used to test until the view is ready.

Miscellaneous Commands

CallHelixVersion

Get the CallHelix version number

Required Parameter List:

none

Optional Parameter List:

none

Reply:

version number (as a 3 digit number)

Syntax Example:

CallHelixVersion

Reply Examples:

212 (integer) -- version 2.1.2

210 (integer) -- version 2.1.0

201 (integer) -- version 2.0.1

Notes:

Because earlier versions of CallHelix do not support this command, attempting to use it with an earlier version produces an error. You can trap this with AppleScript's `try` handler, as in this example:

```
try
  set theVersion to CallHelixVersion -- returns 212 for version 2.1.2
on error
  set theVersion to 100 -- there is no way to distinguish versions prior to 2.0.0
end
```

Get Page Table

Get a collection's internal map.

Required Parameter List:

file specification

Optional Parameter List:

none

Reply:

page table record blocks (a list of integers)

Syntax Example:

get page table theFile

Reply Example:

{1, 35, 1224}

Notes:

get page table retrieves the record block identifiers of the internal page table of the specified collection. This information is useless without detailed knowledge of a Helix collection's internal structure. It is built into CallHelix to provide functionality for utilities written by Autograph Systems (or third parties) with such in depth knowledge.

A collection must be *closed* in order to get its page table.

Appendix B: Helix Apple Event Error Codes

Helix specific errors

When a Helix related Apple event error occurs, Helix returns an error number to CallHelix. CallHelix, in turn, returns the error number along with a text message describing the error to AppleScript. The numbers correspond to the following error messages.

010	Incorrect number of items in the descriptor list
020	No collection opened with named Helix application
030	Incorrect collection name
040	Collection not in User Mode
050	Illegal user name
060	Invalid password for the requested user
070	No such relation exists
080	No such view exists
090	No such view exists for this user
100	String length in list exceeds 255 characters
110	Invalid transaction code
120	View is not compiled
130	View is not in Show Form
140	View is already open
150	Illegal ProcessID sent to test view for readiness
160	Illegal ProcessID sent to close process
170	Only numbers are accepted
180	An end-of-list error was encountered
190	Illegal number of rows requested
200	Tried to store data on a list view
210	Tried to store data on an entry view with no valid fields
220	Store encountered problems with the input data
230	Retrieve setup process was not finished
240	Illegal record ID sent to delete records
250	Delete encountered a record locking problem
260	Maximum number of user logins exceeded
270	Retrieve encountered a locked record (new in Helix 5.0)
1000	An error was encountered trying to store data on an entry view set to stop on errors

CallHelix specific errors

In addition, CallHelix has its own set of errors for problems that it traps directly.

916	Need to supply action: either through 'selector' or 'action'
917	You have exceeded the numbers of rows/processes the demo version can return.
918	You must specify both field and record delimiters, in that order.

Application

Always enclose routines in AppleScript try statements and use on error to handle errors as they occur:

```
tell application "Helix Scripting"
  try
    CallHelix {CollectionName, UserName, UserPassword, RelationName, ViewName} action retrieve records as string
  on error errorText number errorNumber
    display dialog ( errorText & " #" & errorNumber ) as string
  end try
end tell
```

Appendix C: Helix HAEC Resources

The limit on the number of records that can be processed in a single call is controlled by the HAEC (Helix Apple Event Control) resources found in the Helix application. We do not recommend changing these, since no appreciable performance benefit is gained and performance of other clients is diminished. They are documented here only for the sake of completeness.

HAEC ID	Process	Default Value
1	retrieve records	10
2	store records	10
3	delete records	10

Appendix D: CallHelix Demo Version

About the CallHelix Demo version

The demo version of the CallHelix package is fully functional and non-expiring; it works exactly the same as the full version with the following limitations.

1. You can not retrieve more than 10 records via a process.
2. No more than 10 processes can be used before the Helix application must be relaunched.

Demo version distribution

The CallHelix Demo package may be freely distributed, as long as the package is kept complete. You may not distribute the demo components individually.

You can also distribute this URL <http://scripting.autographsystems.com/callhelix/demo.html> which always links to the current demo package.

Upgrading to the full version

If you begin with the demo version and later decide to purchase a CallHelix license, you can upgrade simply by replacing the demo components with the full version. Scripts written for the demo version are fully compatible with the full version.

Appendix E: Targeting a Remote Computer

Introduction

CallHelix can communicate with Helix applications over a network, using Apple Events. OS 9.2.2 and earlier allow this over AppleTalk networks. OS 9.0 and later allows this over TCP/IP networks. Using TCP/IP you can write scripts that modify databases anywhere in the world – as long as the machine it is hosted on has an addressable IP Address and is properly configured.

CallHelix also allows direct access to Helix applications running on remote machines running Mac OS X 10.3 and higher via the optional target parameter. This allows you to access a remote Helix application regardless of whether CallHelix is installed on that machine.

Requirements for remote scripting

To use the direct access capabilities of CallHelix you must use one of the OS X native CallHelix components: *CallHelix.osax* or *Helix Scripting*. You must also be running OS X 10.3 or higher. Classic CallHelix can not be used for this purpose.

It is recommended that you develop exclusively in OS X or in OS 9, as Apple's inter-version Apple Event communication seems to be quite buggy and mixing the two is difficult, at best. This appendix discusses OS X only. OS 9 users should refer to the document "OS 9 Remote Access" in the CallHelix package.

Configuring your computer for remote access

It is beyond the scope of this manual to fully describe how to configure your computers for remote Apple Event access. Essentially it entails opening the "Sharing" System Preference and turning on the "Remote Apple Events" service. If the Firewall is active on the target computer, make sure the appropriate port (3031) is open.

A simple test to see if you have successfully configured your computer for remote access is to send a simple activate command to the Finder:

```
tell application "Finder" of machine "eppc://target_computer"
  activate
end tell
```

If this is successful, the Finder becomes the active application on the target computer.

More information about remote Apple Events can be found at:

http://developer.apple.com/documentation/Carbon/Reference/Apple_Event_Manager/

Targeting Helix on a remote machine

When communicating with Helix on a remote machine, you can target a copy of Helix Scripting running on that machine, or you can target the Helix application directly.

Communication via Helix Scripting

The more straightforward method is to communicate with Helix Scripting on a remote machine. To do this, use the of machine parameter in the tell statement:

```
tell application "Helix Scripting" of machine "eppc://target_computer"
  CallHelixVersion -- sample command. Any CallHelix command can be used here
end tell
```

Communication directly with Helix

You can also communicate directly with Helix running on a remote computer via the optional target parameter:

```
tell application "Helix Scripting"
  CallHelix {"MyColl", "MyUsr", "MyPw", "MyRel", "MyView"} action retrieve records as string target "eppc://target_mac/target_helix"
end tell
```

Note that in this scenario, you must tell your local copy of Helix Scripting to execute the commands.

Machine & Target parameter details

Whether using the of machine or the target command, the parameter string that follows it contains various pieces of information. Some are optional, others are required. The full parameter list is:

```
eppc://[username[:password]@]hostname/appname[[:uid=#]&[:pid=#]]
```

(The bracket characters indicate optional parameters; they are not included in the actual string.) The parameter string breaks down as follows:

eppc: *Required.* The communication protocol. `eppc://` stands for “Event Program to Program Communication” which is Apple’s protocol for communication between programs. The string `eppc://` is required at the beginning of any target statement.

username: *Optional.* The first parameter is the name of a user who has permission to access applications on the remote machine. Typically this is the user who is currently logged in on that machine. If this is not supplied, OS X presents a dialog asking you for a username and password with the required permission.

password: *Optional.* If a username is included in the string, the user’s password can also be included. The password is separated from the username by a “:” as in `username:password`. If this is not supplied, OS X presents a dialog asking you for a username and password with the required permission. If a username was supplied in the parameter string, that username is automatically filled in.

hostname: *Required.* The name of the remote machine. This can be an IP Address (192.168.0.2) or a fully qualified hostname (`myserver.mydomain.com`). On a local network, you can also use the Bonjour name (without the `.local` suffix).

appname: *Required.* The name of the Helix application you want to address. You may find that you have to encode non-UTF-8 characters and white space using standard URL-encoding. (for example, `Helix%20Server`, where the space is replaced with its encoded form: `%20`.)

uid & pid: *Optional.* The `userid` and `processid` assigned by OS X when the application was launched. If a `uid` is supplied, events are sent to the application only if it was launched by that user. If a `pid` is supplied, the `uid` and `appname` are ignored, and the event is delivered only to the application with that `pid`.

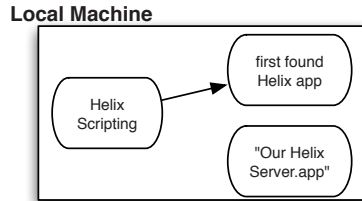
Sample code

Sample code for targeting a remote machine is found in the CallHelix installation package.

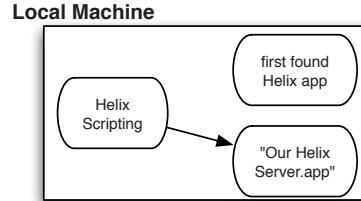
Results of various target statements

This chart shows the results of the various forms of the target parameter, whether targeting the local machine or one on the network.

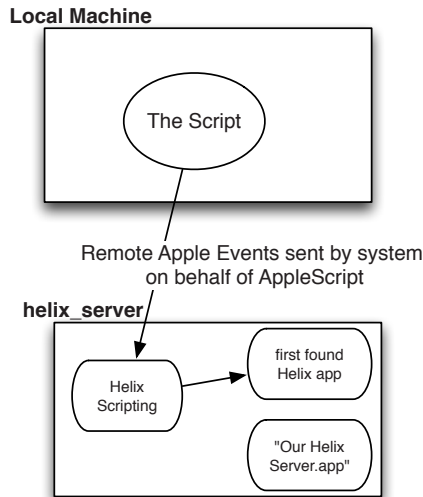
```
tell app "Helix Scripting"
  CallHelix {...} selector ....
```



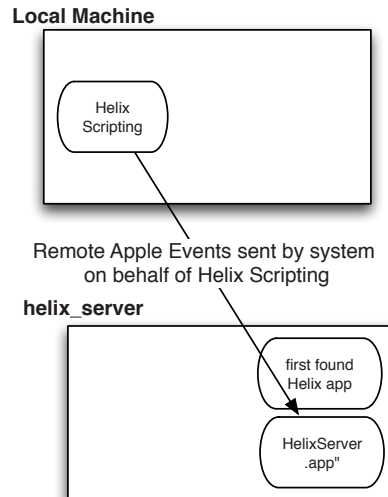
```
tell app "Helix Scripting"
  CallHelix {...} selector .... target -
  "Our Helix Server.app"
```



```
tell app "Helix Scripting" of machine -
  "eppc://helix_server"
  CallHelix {...} selector ....
```



```
tell app "Helix Scripting"
  CallHelix {...} selector .... target -
  "eppc://helix_server/HelixServer.app"
```



Appendix F: Version History

Version 1.0

Initial release by Communications Unlimited

Version 1.1

Interim release by Communications Unlimited

Changes unknown.

Version 1.2

Initial release by Autograph Systems (September, 1999)

Store multiple implemented – previously it dropped into the debugger.

Invalid selectors return an error (110: Invalid Transaction Code) – previously it crashed.

CallHelix Suite ID changed to ChXo to separate CallHelix from Apple Scripting Additions.

Helix RADE, Server, and Runtime are now all supported in a single OSAX. Prior versions required separate additions for each Helix application, and you had to restart when switching from one version to the other.

Dictionary text improved.

Ported to MPW.

Version 1.3

General release (May, 2000)

Check view state support implemented.

Retrieve records, with ErrCheck sample corrected.

Version 1.3.2

Helix 5.0 compatibility release (September 25, 2000)

Get View Data: The error that resulted in the first 4 bytes of data being interpreted as a number when sending a type 0 (rectangle labels) request has been fixed.

Get View Data: The parameter error that resulted when sending a type 1 (view header) request has been fixed.

Fixed a memory leak when retrieving records with a type 2 (“view data”) data request.

Version 2.0

Feature release (July 26, 2004)

PowerPC and OS X native code

Faceless Background Application option

New Commands added:

- get view summary

- retrieve records (as string and as list)

- get view data (as string and as list)

- get partial view data (as list)

- CallHelixVersion

Demo version

Target Helix application parameter added

Integrated Error Code handler

Full Helix 5.x (and later) compatibility

Get View Data: The unsupported “double true” parameter is now documented.

Fixed buffer overflow when trying to access a item in the descriptor list that may, or may not, be there.

Version 2.1

Feature release (January 1, 2006)

Enhanced target parameter allows direct access to remote Helix applications.

Version 2.1.1

Bug fix release (March 8, 2006)

Get Partial View Data: (selector 143) was ignoring the Subform Tabbing parameter.

Corrected a typo in the copyright notice text.

Version 2.1.2

Mac OS X 10.5 Compatibility release (October 13, 2008)

Addressed AppleScript change to Unicode text.

Quick Reference Card

Command	Parameters	Selector #
Storing Data		
store one record	Page 31	110
Required Parameters	{ collection, user, password, relation, view, record data }	
Optional Parameters	target	
Data Returned	{ success/failure flag }	
create process for store	Page 31	111
Required Parameters	{ collection, user, password, relation, view }	
Optional Parameters	target	
Data Returned	ProcessID	
store record(s)	Page 32	112
Required Parameters	{ processID, record data }	
Optional Parameters	{ record data, record data, ... }, target	
Data Returned	{ success/failure flags }	
Retrieving Data		
retrieve records as string	Page 32	104
Required Parameters	{ collection, user, password, relation, view }	
Optional Parameters	with record ids, target	
Data Returned	{ { helix record: "DataRectangle1→DataRect2...", [record id: #] }, { helix record: "DataRect1→DataRect2...", [record id: #] }, ... }	
retrieve records as list	Page 33	105
Required Parameters	{ collection, user, password, relation, view }	
Optional Parameters	with record ids, target	
Data Returned	{ { helix record: { DataRectangle1, DataRectangle2 }, [record id: #] }, { helix record: { DataRect1, DataRect2 }, [record id: #] }, ... }	
create process for retrieve	Page 33	100
Required Parameters	{ collection, user, password, relation, view }	
Optional Parameters	target	
Data Returned	ProcessID	
get view data as ...		
Required Parameters	Data Type 0 or 1: { processID, Data Type } -- Data Type 0: Data Rectangle Names, 1: Outer Abaci Data Data Type 2: { processID, Data Type, StartRecord, NumberToGet }	
Optional Parameters	Data Type 0 or 1: target Data Type 2: { IndentSubforms }, with record ids, delimiters, target	
... as Helix data	Page 33	140
Data Returned	Data Type 0: { { 1, "DataRectangle1Name→DataRectangle2Name..." } } Data Type 1: { { 1, "OuterAbacus1→OuterAbacus2..." } } Data Type 2: { ItemsInList, RecordID, RecordData, RecordID, RecordData, ... }	
... as string	Page 34	141
Data Returned	Data Type 0: { { helix record: "DataRectangle1Name→DataRectangle2Name..." } } Data Type 1: { { helix record: "OuterAbacus1→OuterAbacus2..." } } Data Type 2: { { helix record: "DataRectangle1→DataRectangle2..." [record id: #] }, ... }	
... as list	Page 35	142
Data Returned	Data Type 0: { { helix record: { Field1Name, Field2Name, ... } } } Data Type 1: { { helix record: { OuterAbacus1, OuterAbacus2, ... } } } Data Type 2: { { helix record: { DataRectangle1, DataRectangle2, ... }, [record id: #] }, ... }	
get partial view data	Page 35	143
Required Parameters	{ processID }	
Optional Parameters	{ StartRecord, NumberToGet, IndentSubforms }, with record ids, delimiters, target	
Data Returned	{ { helix record: { DataRectangle1, DataRectangle2, ... }, [record id: #] }, ... }	
Deleting Data		
delete record(s)	Page 36	150
Required Parameters	{ processID, recordID }	
Optional Parameters	{ recordID, recordID, ... }, target	
Data Returned	# of records successfully deleted	
Complex Process Support (Processes w/ProcessID)		
close process	Page 36	130
Required Parameters	{ processID }	
Optional Parameters	target	
Data Returned	<none>	
get view summary	Page 37	121
Required Parameters	{ processID }	
Optional Parameters	target	
Data Returned	{ record count: #, field delimiter: character, record delimiter: character }	
test view for readiness	Page 37	120
Required Parameters	{ processID }	
Optional Parameters	target	
Data Returned	{ false } or { true, RecordsFound, FieldDelimiter, RecordDelimiter }	
check view state	Page 37	190
Required Parameters	{ collection, user, password, relation, view, CheckForList }	
Optional Parameters	target	
Data Returned	error if view is a list and CheckForList is true	
CallHelixVersion	Page 38	n/a
Data Returned	version number as 3 digit integer (eg. 212 = version 2.1.2)	

